1979 ACM Turing Award Lecture

Delivered at ACM '79, Detroit, Oct. 29, 1979

The 1979 ACM Turing Award was presented to Kenneth E. Iverson by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Detroit, Michigan, October 29, 1979.

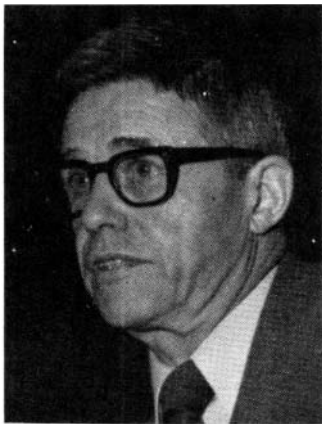
In making its selection, the General Technical Achievement Award Committee cited Iverson for his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL. Iverson's contributions to the implementation of interactive systems, to the educational uses of APL, and to programming language theory and practice were also noted.

Born and raised in Canada, Iverson received his doctorate in 1954 from Harvard University. There he served as Assistant Professor of Applied Mathematics from 1955–1960. He then joined International Business Machines, Corp. and in 1970 was named an IBM Fellow in honor of his contribution to the development of APL.

Dr. Iverson is presently with I.P. Sharp Associates in Toronto. He has published numerous articles on programming languages and has written four books about programming and mathematics: *A Programming Language* (1962), *Elementary Functions* (1966), *Algebra: An Algorithmic Treatment* (1972), and *Elementary Analysis* (1976).

Notation as a Tool of Thought

Kenneth E. Iverson
IBM Thomas J. Watson Research Center



Key Words and Phrases: APL, mathematical notation
CR Category: 4.2

Author's present address: K.E. Iverson, I.P Sharp Associates, 145 King Street West, Toronto, Ontario, Canada M5H1J8.

The importance of nomenclature, notation, and language as tools of thought has long been recognized. In chemistry and in botany, for example, the establishment of systems of nomenclature by Lavoisier and Linnaeus did much to stimulate and to channel later investigation. Concerning language, George Boole in his *Laws of Thought* [1, p.24] asserted "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought. Recognition of the important role of notation in mathematics is clear from the quotations from mathematicians given in Cajori's *A History of Mathematical Notations* [2, pp.332,331]. They are well worth reading in full, but the following excerpts suggest the tone:

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
>
> A.N. Whitehead

> The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid.
>
> Charles Babbage

Nevertheless, mathematical notation has serious deficiencies. In particular, it lacks universality, and must be interpreted differently according to the topic, according to the author, and even according to the immediate context. Programming languages, because they were designed for the purpose of directing computers, offer important advantages as tools of thought. Not only are they universal (general-purpose), but they are also executable and unambiguous. Executability makes it possible to use computers to perform extensive experiments on ideas expressed in a programming language, and the lack of ambiguity makes possible precise thought experiments. In other respects, however, most programming languages are decidedly inferior to mathematical notation and are little used as tools of thought in ways that would be considered significant by, say, an applied mathematician.

The thesis of the present paper is that the advantages of executability and universality found in programming languages can be effectively combined, in a single coherent language, with the advantages offered by mathematical notation. It is developed in four stages:

(a) Section 1 identifies salient characteristics of mathematical notation and uses simple problems to illustrate how these characteristics may be provided in an executable notation.

(b) Sections 2 and 3 continue this illustration by deeper treatment of a set of topics chosen for their general interest and utility. Section 2 concerns polynomials, and Section 3 concerns transformations between representations of functions relevant to a number of topics, including permutations and directed graphs. Although these topics might be characterized as mathematical, they are directly relevant to computer programming, and their relevance will increase as programming continues to develop into a legitimate mathematical discipline.

(c) Section 4 provides examples of identities and formal proofs. Many of these formal proofs concern identities established informally and used in preceeding sections.

(d) The concluding section provides some general comparisons with mathematical notation, references to treatments of other topics, and discussion of the problem of introducing notation in context.

The executable language to be used is APL, a general purpose language which originated in an attempt to provide clear and precise expression in writing and teaching, and which was implemented as a programming language only after several years of use and development [3].

Although many readers will be unfamiliar with APL, I have chosen not to provide a separate introduction to it, but rather to introduce it in context as needed. Mathematical notation is always introduced in this way rather than being taught, as programming languages commonly are, in a separate course. Notation suited as a tool of thought in any topic should permit easy introduction in the context of that topic; one advantage of introducing APL in context here is that the reader may assess the relative difficulty of such introduction.

However, introduction in context is incompatible with complete discussion of all nuances of each bit of notation, and the reader must be prepared to either extend the definitions in obvious and systematic ways as required in later uses, or to consult a reference work. All of the notation used here is summarized in Appendix A, and is covered fully in pages 24-60 of *APL Language* [4].

Readers having access to some machine embodiment of APL may wish to translate the function definitions given here in *direct definition* form [5, p.10] (using ⍺ and ⍵ to represent the left and right arguments) to the *canonical* form required for execution. A function for performing this translation automatically is given in Appendix B.

1. Important Characteristics of Notation

In addition to the executability and universality emphasized in the introduction, a good notation should embody characteristics familiar to any user of mathematical notation:

- Ease of expressing constructs arising in problems.
- Suggestivity.
- Ability to subordinate detail.
- Economy.
- Amenability to formal proofs.

The foregoing is not intended as an exhaustive list, but will be used to shape the subsequent discussion.

Unambiguous executability of the notation introduced remains important, and will be emphasized by displaying below an expression the explicit result produced by it. To maintain the distinction between expressions and results, the expressions will be indented as they automatically are on APL computers. For example, the *integer* function denoted by ⍳ produces a vector of the first ⍵ integers

when applied to the argument N, and the *sum reduction* denoted by $+/$ produces the sum of the elements of its vector argument, and will be shown as follows:

```
      ⍳5
1 2 3 4 5
      +/⍳5
15
```

We will use one non-executable bit of notation: the symbol $\leftrightarrow$ appearing between two expressions asserts their equivalance.

1.1 Ease of Expressing Constructs Arising in Problems

If it is to be effective as a tool of thought, a notation must allow convenient expression not only of notions arising directly from a problem, but also of those arising in subsequent analysis, generalization, and specialization.

Consider, for example, the crystal structure illustrated by Figure 1, in which successive layers of atoms lie not directly on top of one another, but lie "close-packed" between those below them. The numbers of atoms in successive rows from the top in Figure 1 are therefore given by $⍳5$, and the total number is given by $+/⍳5$.

The three-dimensional structure of such a crystal is also close-packed; the atoms in the plane lying above Figure 1 would lie between the atoms in the plane below it, and would have a base row of four atoms. The complete three-dimensional structure corresponding to Figure 1 is therefore a tetrahedron whose planes have bases of lengths 1, 2, 3, 4, and 5. The numbers in successive planes are therefore the *partial* sums of the vector $⍳5$, that is, the sum of the first element, the sum of the first two elements, etc. Such partial sums of a vector V are denoted by $+\backslash V$, the function $+\backslash$ being called *sum scan*. Thus:

```
      +\⍳5
1 3 6 10 15
      +/+\⍳5
35
```

The final expression gives the total number of atoms in the tetrahedron.

The sum $+/⍳5$ can be represented graphically in other ways, such as shown on the left of Figure 2. Combined with the inverted pattern on the right, this representation suggests that the sum may be simply related to the number of units in a rectangle, that is, to a product.

The lengths of the rows of the figure formed by pushing together the two parts of Figure 2 are given by adding the vector $⍳5$ to the same vector reversed. Thus:

```
      ⍳5
1 2 3 4 5
      ⌽⍳5
5 4 3 2 1
      (⍳5)+(⌽⍳5)
6 6 6 6 6
```

Fig. 1.

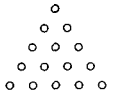
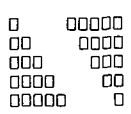


Fig. 2.



This pattern of 5 repetitions of 6 may be expressed as $5⍴6$, and we have:

```
      5⍴6
6 6 6 6 6
      +/5⍴6
30
      6×5
30
```

The fact that $+/5⍴6 \leftrightarrow 6\times5$ follows from the definition of multiplication as repeated addition.

The foregoing suggests that $+/⍳5 \leftrightarrow (6\times5)\div2$, and, more generally, that:

$$+/⍳N \leftrightarrow ((N+1)\times N)\div 2 \qquad \text{A.1}$$

1.2 Suggestivity

A notation will be said to be *suggestive* if the forms of the expressions arising in one set of problems suggest related expressions which find application in other problems. We will now consider related uses of the functions introduced thus far, namely:

$$⍳ \quad ⌽ \quad ⍴ \quad +/ \quad +\backslash$$

The example:

```
      5⍴2
2 2 2 2 2
      ×/5⍴2
32
```

suggests that $\times/M⍴N \leftrightarrow N*M$, where $*$ represents the power function. The similiarity between the definitions of power in terms of times, and of times in terms of plus may therefore be exhibited as follows:

$$\times/M⍴N \leftrightarrow N*M$$
$$+/M⍴N \leftrightarrow N\times M$$

Similar expressions for partial sums and partial products may be developed as follows:

```
      ×\5⍴2
2 4 8 16 32
      2*⍳5
2 4 8 16 32
```

$$\times\backslash M⍴N \leftrightarrow N*⍳M$$
$$+\backslash M⍴N \leftrightarrow N\times⍳M$$

Because they can be represented by a triangle as in Figure 1, the sums $+\backslash⍳5$ are called *triangular* numbers. They are a special case of the *figurate* numbers obtained by repeated applications of sum scan, beginning either with $+\backslash⍳N$, or with $+\backslash N⍴1$. Thus:

```
      5⍴1
1 1 1 1 1
      +\5⍴1
1 2 3 4 5
      +\+\5⍴1
1 3 6 10 15
      +\+\+\5⍴1
1 4 10 20 35
```

Replacing sums over the successive integers by products yields the factorials as follows:

```
      ⍳5
1 2 3 4 5
      ×/⍳5                    ×\⍳5
120                     1 2 6 24 120
      !5                      !⍳5
120                     1 2 6 24 120
```

Part of the suggestive power of a language resides in the ability to represent identities in brief, general, and easily remembered forms. We will illustrate this by expressing *dualities* between functions in a form which embraces DeMorgan's laws, multiplication by the use of logarithms, and other less familiar identities.

If *V* is a vector of positive numbers, then the product `×/V` may be obtained by taking the natural logarithms of each element of *V* (denoted by `⍟V`), summing them (`+/⍟V`), and applying the exponential function (`*+/⍟V`). Thus:

```
×/V ↔ *+/⍟V
```

Since the exponential function `*` is the inverse of the natural logarithm `⍟`, the general form suggested by the right side of the identity is:

```
IG F/G V
```

where *IG* is the function inverse to *G*.

Using `∧` and `∨` to denote the functions *and* and *or*, and `~` to denote the self-inverse function of logical negation, we may express DeMorgan's laws for an arbitrary number of elements by:

```
∧/B ↔ ~∨/~B
∨/B ↔ ~∧/~B
```

The elements of *B* are, of course, restricted to the boolean values `0` and `1`. Using the relation symbols to denote *functions* (for example, `X<Y` yields `1` if *X* is less than *Y* and `0` otherwise) we can express further dualities, such as:

```
≠/B ↔ ~=/~B
=/B ↔ ~≠/~B
```

Finally, using `⌈` and `⌊` to denote the *maximum* and *minimum* functions, we can express dualities which involve arithmetic negation:

```
⌈/V ↔ -⌊/-V
⌊/V ↔ -⌈/-V
```

It may also be noted that scan (`F⍀`) may replace reduction (`F/`) in any of the foregoing dualities.

1.3 Subordination of Detail

As Babbage remarked in the passage cited by Cajori, brevity facilitates reasoning. Brevity is achieved by subordinating detail, and we will here consider three important ways of doing this: the use of arrays, the assignment of names to functions and variables, and the use of operators.

We have already seen examples of the brevity provided by one-dimensional arrays (vectors) in the treatment of duality, and further subordination is provided by matrices and other arrays of higher rank, since functions defined on vectors are extended systematically to arrays of higher rank.

In particular, one may specify the axis to which a function applies. For example, `⌽[1]M` acts along the first axis of a matrix *M* to reverse each of the columns, and `⌽[2]M` reverses each row; `M,[1]N` catenates columns (placing *M* above *N*), and `M,[2]N` catenates rows; and `+/[1]M` sums columns and `+/[2]M` sums rows. If no axis is specified, the function applies along the last axis. Thus `+/M` sums rows. Finally, reduction and scan along the *first* axis may be denoted by the symbols `⌿` and `⍀`.

Two uses of names may be distinguished: *constant* names which have fixed referents are used for entities of very general utility, and ad hoc names are assigned (by means of the symbol `←`) to quantities of interest in a narrower context. For example, the constant (name) `144` has a fixed referent, but the names *CRATE*, *LAYER*, and *ROW* assigned by the expressions

```
CRATE ← 144
LAYER ← CRATE÷8
ROW ← LAYER÷3
```

are ad hoc, or *variable* names. Constant names for vectors are also provided, as in `2 3 5 7 11` for a numeric vector of five elements, and in `'ABCDE'` for a character vector of five elements.

Analogous distinctions are made in the names of functions. Constant names such as `+`, `×`, and `*` are assigned to so-called *primitive* functions of general utility. The detailed definitions, such as `+/M⍴N` for `N×M` and `×/M⍴N` for `N*M`, are subordinated by the constant names `×` and `*`.

Less familiar examples of constant function names are provided by the comma which *catenates* its arguments as illustrated by:

```
(⍳5),(⌽5) ↔ 1 2 3 4 5 5 4 3 2 1
```

and by the *base-representation* function `⊤`, which produces a representation of its right argument in the radix specified by its left argument. For example:

```
      2 2 2 ⊤ 3 ↔ 0 1 1

      2 2 2 ⊤ 4 ↔ 1 0 0

      BN←2 2 2 ⊤ 0 1 2 3 4 5 6 7
      BN
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1

      BN,⌽BN
0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0
0 0 1 1 0 0 1 1 1 1 0 0 1 1 0 0
0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1
```

The matrix *BN* is an important one, since it can be viewed in several ways. In addition to representing the binary numbers, the columns represent all subsets of a set of three elements, as well as the en-

tries in a truth table for three boolean arguments. The general expression for N elements is easily seen to be $(N\rho 2)\tau(\iota 2*N)-1$, and we may wish to assign an ad hoc name to this function. Using the direct definition form (Appendix B), the name $\underline{T}$ is assigned to this function as follows:

```
T:(ωρ2)τ(ι2*ω)-1                                    A.2
```

The symbol ω represents the argument of the function; in the case of two arguments the left is represented by α. Following such a definition of the function $\underline{T}$, the expression $\underline{T}\ 3$ yields the boolean matrix BN shown above.

Three expressions, separated by colons, are also used to define a function as follows: the middle expression is executed first; if its value is zero the first expression is executed, if not, the last expression is executed. This form is convenient for recursive definitions, in which the function is used in its own definition. For example, a function which produces binomial coefficients of an order specified by its argument may be defined recursively as follows:

```
BC:(X,0)+(0,X←BC ω-1):ω=0:1                         A.3
```

Thus $BC\ 0 \leftrightarrow 1$ and $BC\ 1 \leftrightarrow 1\ 1$ and $BC\ 4 \leftrightarrow 1\ 4\ 6\ 4\ 1$.

The term *operator*, used in the strict sense defined in mathematics rather than loosely as a synonym for *function*, refers to an entity which applies to functions to produce functions; an example is the derivative operator.

We have already met two operators, *reduction*, and *scan*, denoted by $/$ and $\backslash$, and seen how they contribute to brevity by applying to different functions to produce families of related functions such as $+/$ and $\times/$ and $\wedge/$. We will now illustrate the notion further by introducing the *inner product* operator denoted by a period. A function (such as $+/$) produced by an operator will be called a *derived* function.

If P and Q are two vectors, then the inner product $+.\times$ is defined by:

```
P+.×Q ↔ +/P×Q
```

and analogous definitions hold for function pairs other than $+$ and $\times$. For example:

```
      P←2 3 5
      Q←2 1 2
      P+.×Q
17
      P×.*Q
300
      P⌊.+Q
4
```

Each of the foregoing expressions has at least one useful interpretation: $P+.\times Q$ is the total cost of order quantities Q for items whose prices are given by P; because P is a vector of primes, $P\times.*Q$ is the number whose prime decomposition is given by the exponents Q; and if P gives distances from a source to transhipment points and Q gives distances from the transhipment points to the destination, then $P\lfloor.+Q$ gives the minimum distance possible.

The function $+.\times$ is equivalent to the inner product or dot product of mathematics, and is extended to matrices as in mathematics. Other cases such as $\times.*$ are extended analogously. For example, if $\underline{T}$ is the function defined by A.2, then:

```
      T 3
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
      P+.×T 3
0 5 3 8 2 7 5 10
      P×.*T 3
1 5 3 15 2 10 6 30
```

These examples bring out an important point: if B is boolean, then $P+.\times B$ produces sums over subsets of P specified by 1's in B, and $P\times.*B$ produces products over subsets.

The phrase $\circ.\times$ is a special use of the inner product operator to produce a derived function which yields products of each element of its left argument with each element of its right. For example:

```
      2 3 5∘.×ι5
2  4  6  8 10
3  6  9 12 15
5 10 15 20 25
```

The function $\circ.\times$ is called *outer product*, as it is in tensor analysis, and functions such as $\circ.+$ and $\circ.*$ and $\circ.<$ are defined analogously, producing "function tables" for the particular functions. For example:

```
      D←0 1 2 3
      D∘.⌈D
0 1 2 3
1 1 2 3
2 2 2 3
3 3 3 3
      D∘.≥D
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
      D∘.!D
1 1 1 1
0 1 2 3
0 0 1 3
0 0 0 1
```

The symbol ! denotes the binomial coefficient function, and the table $D\circ.!D$ is seen to contain Pascal's triangle with its apex at the left; if extended to negative arguments (as with $D\leftarrow{}^-3\ {}^-2\ {}^-1\ 0\ 1\ 2\ 3$) it will be seen to contain the triangular and higher-order figurate numbers as well. This extension to negative arguments is interesting for other functions as well. For example, the table $D\circ.\times D$ consists of four quadrants separated by a row and a column of zeros, the quadrants showing clearly the rule of signs for multiplication.

Patterns in these function tables exhibit other properties of the functions, allowing brief statements of proofs by exhaustion. For example, commutativity appears as a symmetry about the diagonal. More precisely, if the result of the transpose function $\Diamond$ (which reverses the order of the axes of its argument) applied to a table $T\leftarrow D\circ.\lceil D$ agrees with T, then the function $\lceil$ is commutative on the domain. For example, $T=\Diamond T\leftarrow D\circ.\lceil D$ produces a table of 1's because $\lceil$ is commutative.

Corresponding tests of associativity require rank 3 tables of the form `D∘.f(D∘.fD)` and `(D∘.fD)∘.fD`. For example:

```
      D←0 1
D∘.∧(D∘.∧D)   (D∘.∧D)∘.∧D   D∘.≤(D∘.≤D)   (D∘.≤D)∘.≤D

 0 0           0 0           1 1           0 1
 0 0           0 0           1 1           0 1

 0 0           0 0           1 1           1 1
 0 1           0 1           0 1           0 1
```

1.4 Economy

The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important.

Economy requires that a large number of ideas be expressible in terms of a relatively small vocabulary. A fundamental scheme for achieving this is the introduction of grammatical rules by which meaningful phrases and sentences can be constructed by combining elements of the vocabulary.

This scheme may be illustrated by the first example treated -- the relatively simple and widely useful notion of the sum of the first `N` integers was not introduced as a primitive, but as a phrase constructed from two more generally useful notions, the function `⍳` for the production of a vector of integers, and the function `+/` for the summation of the elements of a vector. Moreover, the derived function `+/` is itself a phrase, summation being a derived function constructed from the more general notion of the reduction operator applied to a particular function.

Economy is also achieved by generality in the functions introduced. For example, the definition of the factorial function denoted by `!` is not restricted to integers, and the gamma function of `X` may therefore be written as `!X-1`. Similiarly, the *relations* defined on all real arguments provide several important logical functions when applied to boolean arguments: exclusive-or (`≠`), material implication (`≤`), and equivalence (`=`).

The economy achieved for the matters treated thus far can be assessed by recalling the vocabulary introduced:

```
⍳  ⍴  ⍉  ⍋  ,
/  \  .

+-×÷*⍟!⌈⌊⍱
∨∧~<≤=≥>≠
```

The five functions and three operators listed in the first two rows are of primary interest, the remaining familiar functions having been introduced to illustrate the versatility of the operators.

A significant economy of symbols, as opposed to economy of functions, is attained by allowing any symbol to represent both a *monadic* function (i.e. a function of one argument) and a *dyadic* function, in the same manner that the minus sign is commonly used for both subtraction and negation. Because the two functions represented may, as in the case of the minus sign, be related, the burden of remembering symbols is eased.

For example, `X*Y` and `*Y` represent power and exponential, `X⍟Y` and `⍟Y` represent base `X` logarithm and natural logarithm, `X÷Y` and `÷Y` represent division and reciprocal, and `X!Y` and `!Y` represent the binomial coefficient function and the factorial (that is, `X!Y←→(!Y)÷(!X)×(!Y-X)`). The symbol `⍴` used for the dyadic function of replication also represents a monadic function which gives the shape of the argument (that is, `X←→⍴X⍴Y`), the symbol `⌽` used for the monadic reversal function also represents the dyadic *rotate* function exemplified by `2⌽⍳5←→3 4 5 1 2`, and by `¯2⌽⍳5←→4 5 1 2 3`, and finally, the comma represents not only catenation, but also the monadic *ravel*, which produces a vector of the elements of its argument in "row-major" order. For example:

```
      T←2                ,T
0 0 1 1          0 0 1 1 0 1 0 1
0 1 0 1
```

Simplicity of the grammatical rules of a notation is also important. Because the rules used thus far have been those familiar in mathematical notation, they have not been made explicit, but two simplifications in the order of execution should be remarked:

(1) All functions are treated alike, and there are no rules of precedence such as `×` being executed before `+`.

(2) The rule that the right argument of a monadic function is the value of the entire expression to its right, implicit in the order of execution of an expression such as `SIN LOG !N`, is extended to dyadic functions.

The second rule has certain useful consequences in reduction and scan. Since `F/V` is equivalent to placing the function `F` between the elements of `V`, the expression `-/V` gives the alternating sum of the elements of `V`, and `÷/V` gives the alternating product. Moreover, if `B` is a boolean vector, then `<\B` "isolates" the first `1` in `B`, since all elements following it become `0`. For example:

```
<\0 0 1 1 0 1 1 ←→ 0 0 1 0 0 0 0
```

Syntactic rules are further simplified by adopting a single form for all dyadic functions, which appear between their arguments, and for all monadic functions, which appear before their arguments. This contrasts with the variety of rules in mathematics. For example, the symbols for the monadic functions of negation, factorial, and mag-

nitude precede, follow, and surround their arguments, respectively. Dyadic functions show even more variety.

1.5 Amenability to Formal Proofs

The importance of formal proofs and derivations is clear from their role in mathematics. Section 4 is largely devoted to formal proofs, and we will limit the discussion here to the introduction of the forms used.

Proof by exhaustion consists of exhaustively examining all of a finite number of special cases. Such exhaustion can often be simply expressed by applying some outer product to arguments which include all elements of the relevant domain. For example, if `D←0 1`, then `D∘.∧D` gives all cases of application of the *and* function. Moreover, DeMorgan's law can be proved exhaustively by comparing each element of the matrix `D∘.∧D` with each element of `~(~D)∘.∨(~D)` as follows:

```
      D∘.∧D                    ~(~D)∘.∨(~D)
 0 0                      0 0
 0 1                      0 1
      (D∘.∧D)=(~(~D)∘.∨(~D))
 1 1
 1 1
      ∧/,(D∘.∧D)=(~(~D)∘.∨(~D))
1
```

Questions of associativity can be addressed similarly, the following expressions showing the associativity of *and* and the non-associativity of *not-and*:

```
      ∧/,((D∘.∧D)∘.∧D)=(D∘.∧(D∘.∧D))
1
      ∧/,((D∘.⍲D)∘.⍲D)=(D∘.⍲(D∘.⍲D))
0
```

A proof by a sequence of identities is presented by listing a sequence of expressions, annotating each expression with the supporting evidence for its equivalence with its predecessor. For example, a formal proof of the identity A.1 suggested by the first example treated would be presented as follows:

```
+/⍳N
+/⌽⍳N                                  + is associative and commutative
((+/⍳N)+(+/⌽⍳N))÷2                                        (X+X)÷2←→X
(+/(⍳N)+(⌽⍳N))÷2                       + is associative and commutative
(+/((N+1)⍴N))÷2                                                 Lemma
((N+1)×N)÷2                                          Definition of ×
```

The fourth annotation above concerns an identity which, after observation of the pattern in the special case `(⍳5)+(⌽⍳5)`, might be considered obvious or might be considered worthy of formal proof in a separate lemma.

Inductive proofs proceed in two steps: 1) some identity (called the *induction hypothesis*) is assumed true for a fixed integer value of some parameter `N` and this assumption is used to prove that the identity also holds for the value `N+1`, and 2) the identity is shown to hold for some integer value `K`. The conclusion is that the identity holds for all integer values of `N` which equal or exceed `K`.

Recursive definitions often provide convenient bases for inductive proofs. As an example we will use the recursive definition of the binomial coefficient function `BC` given by A.3 in an inductive proof showing that the sum of the binomial coefficients of order `N` is `2*N`. As the induction hypothesis we assume the identity:

```
+/BC N ←→ 2*N
```

and proceed as follows:

```
+/BC N+1
+/(X,0)+(0,X←BC N)                                                  A.3
(+/X,0)+(+/0,X)          + is associative and commutative
(+/X)+(+/X)                                                  0+Y←→Y
2×+/X                                                      Y+Y←→2×Y
2×+/BC N                                                Definition of X
2×2*N                                                 Induction hypothesis
2*N+1                                                 Property of Power (*)
```

It remains to show that the induction hypothesis is true for some integer value of `N`. From the recursive definition A.3, the value of `BC 0` is the value of the rightmost expression, namely `1`. Consequently, `+/BC 0` is `1`, and therefore equals `2*0`.

We will conclude with a proof that DeMorgan's law for scalar arguments, represented by:

```
A∧B ←→ ~(~A)∨(~B)                                                   A.4
```

and proved by exhaustion, can indeed be extended to vectors of arbitrary length as indicated earlier by the putative identity:

```
∧/V ←→ ~∨/~V                                                        A.5
```

As the induction hypothesis we will assume that A.5 is true for vectors of length `(⍴V)-1`.

We will first give formal recursive definitions of the derived functions *and*-reduction and *or*-reduction (`∧/` and `∨/`), using two new primitives, *indexing*, and *drop*. Indexing is denoted by an expression of the form `X[I]`, where `I` is a single index or array of indices of the vector `X`. For example, if `X←2 3 5 7`, then `X[2]` is `3`, and `X[2 1]` is `3 2`. Drop is denoted by `K↓X` and is defined to drop `|K` (i.e., the magnitude of `K`) elements from `X`, from the head if `K>0` and from the tail if `K<0`. For example, `2↓X` is `5 7` and `¯2↓X` is `2 3`. The *take* function (to be used later) is denoted by `↑` and is defined analogously. For example, `3↑X` is `2 3 5` and `¯3↑X` is `3 5 7`.

The following functions provide formal definitions of *and*-reduction and *or*-reduction:

```
ANDRED:⍵[1]∧ANDRED 1↓⍵:0=⍴⍵:1                                       A.6
ORRED :⍵[1]∨ ORRED 1↓⍵:0=⍴⍵:0                                       A.7
```

The inductive proof of A.5 proceeds as follows:

```
∧/V
(V[1])∧(∧/1↓V)                                                      A.6
~(~V[1])∨(~∧/1↓V)                                                   A.4
~(~V[1])∨(~~∨/~1↓V)                                                 A.5
~(~V[1])∨(∨/~1↓V)                                            ~~X←→X
~∨/(~V[1]),(~1↓V)                                                   A.7
~∨/~(V[1],1↓V)                                  ∨ distributes over ,
~∨/~V                                  Definition of , (catenation)
```

2. Polynomials

If C is a vector of coefficients and X is a scalar, then the polynomial in X with coefficients C may be written simply as `+/C×X*¯1+⍳⍴C`, or `+/(X*¯1+⍳⍴C)×C`, or `(X*¯1+⍳⍴C)+.×C`. However, to apply to a non-scalar array of arguments X, the power function `*` should be replaced by the power table `∘.*` as shown in the following definition of the polynomial function:

```
P:(ω∘.*¯1+⍳⍴α)+.×α                                    B.1
```

For example, `1 3 3 1 P 0 1 2 3 4 ↔ 1 8 27 64 125`. If `⍴α` is replaced by `1+⍴α`, then the function applies also to matrices and higher dimensional arrays of sets of coefficients representing (along the leading axis of `α`) collections of coefficients of different polynomials.

This definition shows clearly that the polynomial is a linear function of the coefficient vector. Moreover, if `α` and `ω` are vectors of the same shape, then the pre-multiplier `ω∘.*¯1+⍳⍴α` is the Vandermonde matrix of `ω` and is therefore invertible if the elements of `ω` are distinct. Hence if C and X are vectors of the same shape, and if `Y←C P X`, then the inverse (curve-fitting) problem is clearly solved by applying the matrix inverse function `⌹` to the Vandermonde matrix and using the identity:

```
C ↔ (⌹X∘.*¯1+⍳⍴X)+.×Y
```

2.1 Products of Polynomials

The "product of two polynomials B and C" is commonly taken to mean the coefficient vector D such that:

```
D P X ↔ (B P X)×(C P X)
```

It is well-known that D can be computed by taking products over all pairs of elements from B and C and summing over subsets of these products associated with the same exponent in the result. These products occur in the function table `B∘.×C`, and it is easy to show informally that the powers of X associated with the elements of `B∘.×C` are given by the addition table `E←(¯1+⍳⍴B)∘.+(¯1+⍳⍴C)`. For example:

```
      X←2
      B←3 1 2 3
      C←2 0 3
      E←(¯1+⍳⍴B)∘.+(¯1+⍳⍴C)
      B∘.×C           E             X*E
 6 0 9           0 1 2          1  2  4
 2 0 3           1 2 3          2  4  8
 4 0 6           2 3 4          4  8 16
 6 0 9           3 4 5          8 16 32
      +/,(B∘.×C)×X*E
518
      (B P X)×(C P X)
518
```

The foregoing suggests the following identity, which will be established formally in Section 4:

```
(B P X)×(C P X)↔+/,(B∘.×C)×X*(¯1+⍳⍴B)∘.+(¯1+⍳⍴C)      B.2
```

Moreover, the pattern of the exponent table E shows that elements of `B∘.×C` lying on diagonals are associated with the same power, and that the coefficient vector of the product polynomial is therefore given by sums over these diagonals. The table `B∘.×C` therefore provides an excellent organization for the manual computation of products of polynomials. In the present example these sums give the vector `D←6 2 13 9 6 9`, and `D P X` may be seen to equal `(B P X)×(C P X)`.

Sums over the required diagonals of `B∘×C` can also be obtained by bordering it by zeros, skewing the result by rotating successive rows by successive integers, and then summing the columns. We thus obtain a definition for the polynomial product function as follows:

```
PP:+/(1-⍳⍴α)⌽α∘.×ω,1↑0×α
```

We will now develop an alternative method based upon the simple observation that if `B PP C` produces the product of polynomials B and C, then `PP` is linear in both of its arguments. Consequently,

```
PP:α+.×A+.×ω
```

where A is an array to be determined. A must be of rank 3, and must depend on the exponents of the left argument `(¯1+⍳⍴α)`, of the result `(¯1+⍳⍴1+α,ω)`, and of the right argument. The "deficiencies" of the right exponent are given by the difference table `(⍳⍴1+α,ω)∘.-⍳⍴ω`, and comparison of these values with the left exponents yields A. Thus

```
A←(¯1+⍳⍴α)∘.=((⍳⍴1+α,ω)∘.-⍳⍴ω)
```

and

```
PP:α+.×((¯1+⍳⍴α)∘.=(⍳⍴1+α,ω)∘.-⍳⍴ω)+.×ω
```

Since `α+.×A` is a matrix, this formulation suggests that if `D←B PP C`, then C might be obtained from D by pre-multiplying it by the inverse matrix `(⌹B+.×A)`, thus providing division of polynomials. Since `B+.×A` is not square (having more rows than columns), this will not work, but by replacing `M←B+.×A` by either its leading square part `(2⍴⌊/⍴M)↑M`, or by its trailing square part `(-2⍴⌊/⍴M)↑M`, one obtains two results, one corresponding to division with low-order remainder terms, and the other to division with high-order remainder terms.

2.2 Derivative of a Polynomial

Since the derivative of `X*N` is `N×X*N-1`, we may use the rules for the derivative of a sum of functions and of a product of a function with a constant, to show that the derivative of the polynomial `C P X` is the polynomial `(1↓C×¯1+⍳⍴C) P X`. Using this result it is clear that the integral is the polynomial `(A,C÷⍳⍴C) P X`, where A is an arbitrary scalar constant. The expression `1⌽C×¯1+⍳⍴C` also yields the

coefficients of the derivative, but as a vector of the same shape as C and having a final zero element.

2.3 Derivative of a Polynomial with Respect to Its Roots

If R is a vector of three elements, then the derivatives of the polynomial ×/X-R with respect to each of its three roots are -(X-R[2])×(X-R[3]), and -(X-R[1])×(X-R[3]), and -(X-R[1])×(X-R[2]). More generally, the derivative of ×/X-R with respect to R[J] is simply -(X-R)×.*J≠ιρR, and the vector of derivatives with respect to each of the roots is -(X-R)×.*I∘.≠I←ιρR.

The expression ×/X-R for a polynomial with roots R applies only to a scalar X, the more general expression being ×/X∘.-R. Consequently, the general expression for the matrix of derivatives (of the polynomial evaluated at X[I] with respect to root R[J]) is given by:

```
-(X∘.-R)×.*I∘.≠I←ιρR                                    B.3
```

2.4 Expansion of a Polynomial

Binomial expansion concerns the development of an identity in the form of a polynomial in X for the expression (X+Y)*N. For the special case of Y=1 we have the well-known expression in terms of the binomial coefficients of order N:

```
(X+1)*N ↔ ((0,ιN)!N)⍤1 X
```

By extension we speak of the expansion of a polynomial as a matter of determining coefficients D such that:

```
C ⍤1 X+Y ↔ D ⍤1 X
```

The coefficients D are, in general, functions of Y. If Y=1 they again depend only on binomial coefficients, but in this case on the several binomial coefficients of various orders, specifically on the matrix J∘.!J←¯1+ιρC.

For example, if C←3 1 2 4, and C ⍤1 X+1↔D ⍤1 X, then D depends on the matrix:

```
      0 1 2 3 ∘.! 0 1 2 3
1 1 1 1
0 1 2 3
0 0 1 3
0 0 0 1
```

and D must clearly be a weighted sum of the columns, the weights being the elements of C. Thus:

```
D←(J∘.!J←¯1+ιρC)+.×C
```

Jotting down the matrix of coefficients and performing the indicated matrix product provides a quick and reliable way to organize the otherwise messy manual calculation of expansions.

If B is the appropriate matrix of binomial coefficients, then D←B+.×C, and the expansion function is clearly linear in the coefficients C. Moreover, expansion for Y=¯1 must be given by the inverse matrix ⌹B, which will be seen to contain the alternating binomial coefficients. Finally, since:

```
C ⍤1 X+(K+1) ↔ C ⍤1 (X+K)+1 ↔ (B+.×C) ⍤1 (X+K)
```

it follows that the expansion for positive integer values of Y must be given by products of the form:

```
B+.×B+.×B+.×B+.×C
```

where the B occurs Y times.

Because +.× is associative, the foregoing can be written as M+.×C, where M is the product of Y occurrences of B. It is interesting to examine the successive powers of B, computed either manually or by machine execution of the following inner product power function:

```
IPP:α+.×α IPP ω-1:ω=0:J∘.=J←¯1+ι1↑ρα
```

Comparison of B IPP K with B for a few values of K shows an obvious pattern which may be expressed as:

```
B IPP K ↔ B×K*0⌈-J∘.-J←¯1+ι1↑ρB
```

The interesting thing is that the right side of this identity is meaningful for non-integer values of K, and, in fact, provides the desired expression for the general expansion C ⍤1 X+Y:

```
C ⍤1(X+Y) ↔ (((J∘.!J)×Y*0⌈-J∘.-J←¯1+ιρC)+.×C)⍤1 X          B.4
```

The right side of B.4 is of the form (M+.×C)⍤1 X, where M itself is of the form B×Y*E and can be displayed informally (for the case 4=ρC) as follows:

```
1 1 1 1              0 1 2 3
0 1 2 3              0 0 1 2
0 0 1 3     ×Y*      0 0 0 1
0 0 0 1              0 0 0 0
```

Since Y*K multiplies the single-diagonal matrix B×(K=E), the expression for M can also be written as the inner product (Y*J)+.×T, where T is a rank 3 array whose Kth plane is the matrix B×(K=E). Such a rank three array can be formed from an upper triangular matrix M by making a rank 3 array whose first plane is M (that is, (1=ι1↑ρM)∘.×M) and rotating it along the first axis by the matrix J∘-J, whose Kth superdiagonal has the value -K. Thus:

```
DS:(I∘.-I)⌽[1](1=I←ι1↑ρω)∘.×ω                           B.5

      DS  K∘.!K←¯1+ι3
1 0 0
0 1 0
0 0 1

0 1 0
0 0 2
0 0 0

0 0 1
0 0 0
0 0 0
```

Substituting these results in B.4 and using the associativity of +.×, we have the following identity for the expansion of a polynomial, valid for non-integer as well as integer values of Y:

```
C ⍤1 X+Y ↔ ((Y*J)+.×(DS J∘.!J←¯1+ιρC)+.×C)⍤1 X          B.6
```

For example:

```
      Y←3
      C←3 1 4 2
      M←(Y*J)+.×DS J∘.!J←¯1+⍳⍴C
      M
 1  3  9 27
 0  1  6 27
 0  0  1  9
 0  0  0  1
      M+.×C
96 79 22 2
      (M+.×C) ⊥ X←2
358
      C ⊥ X+Y
358
```

3. Representations

The subjects of mathematical analysis and computation can be *represented* in a variety of ways, and each representation may possess particular advantages. For example, a positive integer N may be represented simply by N check-marks; less simply, but more compactly, in Roman numerals; even less simply, but more conveniently for the performance of addition and multiplication, in the decimal system; and less familiarly, but more conveniently for the computation of the least common multiple and the greatest common divisor, in the prime decomposition scheme to be discussed here.

Graphs, which concern connections among a collection of elements, are an example of a more complex entity which possesses several useful representations. For example, a simple directed graph of N elements (usually called *nodes*) may be represented by an N by N boolean matrix B (usually called an *adjacency* matrix) such that B[I;J]=1 if there is a connection *from* node I *to* node J. Each connection represented by a 1 in B is called an *edge*, and the graph can also be represented by a +/,B by N matrix in which each row shows the nodes connected by a particular edge.

Functions also admit different useful representations. For example, a permutation function, which yields a reordering of the elements of its vector argument X, may be represented by a *permutation vector* P such that the permutation function is simply X[P], by a *cycle* representation which presents the structure of the function more directly, by the boolean matrix B←P∘.=⍳⍴P such that the permutation function is B+.×X, or by a *radix* representation R which employs one of the columns of the matrix 1+(⍳N)⊤¯1+⍳!N←⍴X, and has the property that 2|+/R-1 is the parity of the permutation represented.

In order to use different representations conveniently, it is important to be able to express the transformations between representations clearly and precisely. Conventional mathematical notation is often deficient in this respect, and the present section is devoted to developing expressions for the transformations between representations useful in a variety of topics: number systems, polynomials, permutations, graphs, and boolean algebra.

3.1 Number Systems

We will begin the discussion of representations with a familiar example, the use of different representations of positive integers and the transformations between them. Instead of the *positional* or *base-value* representations commonly treated, we will use *prime decomposition*, a representation whose interesting properties make it useful in introducing the idea of logarithms as well as that of number representation [6, Ch.16].

If P is a vector of the first ⍴P primes and E is a vector of non-negative integers, then E can be used to represent the number P×.*E, and all of the integers ⍳⌈/P can be so represented. For example, 2 3 5 7 ×.* 0 0 0 0 is 1 and 2 3 5 7 ×.* 1 1 0 0 is 6 and:

```
      P
2 3 5 7
      ME
 0 1 0 2 0 1 0 3 0 1
 0 0 1 0 0 1 0 0 2 0
 0 0 0 0 1 0 0 0 0 1
 0 0 0 0 0 0 1 0 0 0
      P×.*ME
1 2 3 4 5 6 7 8 9 10
```

The similarity to logarithms can be seen in the identity:

```
      ×/P×.*ME ↔ P×.*+/ME
```

which may be used to effect multiplication by addition.

Moreover, if we define GCD and LCM to give the greatest common divisor and least common multiple of elements of vector arguments, then:

```
      GCD P×.*ME ↔ P×.*⌊/ME
      LCM P×.*ME ↔ P×.*⌈/ME

      ME          V←P×.*ME
2 1 0             V
3 1 2       18900 7350 3087
2 2 0             GCD V                LCM V
1 2 3       21                   926100
                  P×.*⌊/ME             P×.*⌈/ME
            21                   926100
```

In defining the function GCD, we will use the operator / with a boolean argument B (as in B/). It produces the *compression* function which selects elements from its right argument according to the *ones* in B. For example, 1 0 1 0 1/⍳5 is 1 3 5. Moreover, the function B/ applied to a matrix argument compresses rows (thus selecting certain columns), and the function B⌿ compresses columns to select rows. Thus:

```
      GCD:GCD M,(M←⌊/R)|R:1≥⍴R←(ω≠0)/ω:+/R
      LCM:(×/X)÷GCD X←(1↓ω),LCM 1↓ω:0=⍴ω:1
```

The transformation to the value of a number from its prime decomposition representation (VFR) and the inverse transformation to the representation from the value (RFV) are given by:

```
      VFR:α×.*ω
      RFV:D←α RFV ω÷α×.*D:∧/~D←0=α|ω:D
```

For example:

```
      P VFR 2 1 3 1
10500
      P RFV 10500
2 1 3 1
```

3.2 Polynomials

Section 2 introduced two representations of a polynomial on a scalar argument X, the first in terms of a vector of coefficients C (that is, $+/C\times X*{}^{-}1+\iota\rho C$), and the second in terms of its roots R (that is, $\times/X-R$). The coefficient representation is convenient for adding polynomials ($C+D$) and for obtaining derivatives ($1\downarrow C\times{}^{-}1+\iota\rho C$). The root representation is convenient for other purposes, including multiplication which is given by $R1,R2$.

We will now develop a function *CFR* (Coefficients from Roots) which transforms a roots representation to an equivalent coefficient representation, and an inverse function *RFC*. The development will be informal; a formal derivation of *CFR* appears in Section 4.

The expression for *CFR* will be based on Newton's symmetric functions, which yield the coefficients as sums over certain of the products over all subsets of the arithmetic negation (that is, $-R$) of the roots R. For example, the coefficient of the constant term is given by $\times/-R$, the product over the entire set, and the coefficient of the next term is a sum of the products over the elements of $-R$ taken $(\rho R)-1$ at a time.

The function defined by A.2 can be used to give the products over all subsets as follows:

```
      P←(-R)×.*M←⍉ ⍴R
```

The elements of P summed to produce a given coefficient depend upon the number of elements of R excluded from the particular product, that is, upon $+/\sim M$, the sum of the columns of the complement of the boolean "subset" matrix $⍉\rho R$.

The summation over P may therefore be expressed as $((0,\iota\rho R)\circ.=+/\sim M)+.\times P$, and the complete expression for the coefficients C becomes:

```
      C←((0,⍳⍴R)∘.=+/~M)+.×(-R)×.*M←⍉ ⍴R
```

For example, if $R\leftarrow 2\ 3\ 5$, then

```
      M
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
      (-R)×.*M
1 ¯5 ¯3 15 ¯2 10 6 ¯30
      +/~M
3 2 2 1 2 1 1 0
      (0,⍳⍴R)∘.=+/~M
0 0 0 0 0 0 0 1
0 0 0 1 0 1 1 0
0 1 1 0 1 0 0 0
1 0 0 0 0 0 0 0
      ((0,⍳⍴R)∘.=+/~M)+.×(-R)×.*M←⍉ ⍴R
¯30 31 ¯10 1
```

The function *CFR* which produces the coefficients from the roots may therefore be defined and used as follows:

```
      CFR:((0,⍳⍴ω)∘.=+/~M)+.×(-ω)×.*M←⍉ ⍴ω          C.1

      CFR 2 3 5
¯30 31 ¯10 1
      (CFR 2 3 5) ⍉ X←1 2 3 4 5 6 7 8
¯8 0 0 ¯2 0 12 40 90
      ×/X∘.-2 3 5
¯8 0 0 ¯2 0 12 40 90
```

The inverse transformation *RFC* is more difficult, but can be expressed as a successive approximation scheme as follows:

```
RFC:(¯1+⍳⍴1+ω)G ω
G:(α-Z)G ω:TOL≥⌈/|Z←α STEP ω:α-Z
STEP:(⊞(α∘.-α)×.*I∘.≠I←⍳⍴α)+.×(α∘.*¯1+⍳⍴ω)+.×ω

      □←C←CFR 2 3 5 7
210 ¯247 101 ¯17 1
      TOL←1E¯8
      RFC C
7 5 2 3
```

The order of the roots in the result is, of course, immaterial. The final element of any argument of *RFC* must be 1, since any polynomial equivalent to $\times/X-R$ must necessarily have a coefficient of 1 for the high order term.

The foregoing definition of *RFC* applies only to coefficients of polynomials whose roots are all real. The left argument of G in *RFC* provides (usually satisfactory) initial approximations to the roots, but in the general case some at least must be complex. The following example, using the roots of unity as the initial approximation, was executed on an APL system which handles complex numbers:

```
      (*○0J2×(¯1+⍳N)÷N←⍴1+ω)Gω                      C.2

      □←C←CFR 1J1 1J¯1 1J2 1J¯2
10 ¯14 11 ¯4 1
      RFC C
1J¯1 1J2 1J1 1J¯2
```

The monadic function ○ used above multiplies its argument by pi.

In Newton's method for the root of a scalar function F, the next approximation is given by $A\leftarrow A-(F\ A)\div DF\ A$, where *DF* is the derivative of F. The function *STEP* is the generalization of Newton's method to the case where F is a vector function of a vector. It is of the form $(⊞M)+.\times B$, where B is the value of the polynomial with coefficients ω, the original argument of *RFC*, evaluated at α, the current approximation to the roots; analysis similar to that used to derive B.3 shows that M is the matrix of derivatives of a polynomial with roots α, the derivatives being evaluated at α.

Examination of the expression for M shows that its off-diagonal elements are all zero, and the expression $(⊞M)+.\times B$ may therefore be replaced by $B\div D$, where D is the vector of diagonal elements of M. Since $(I,J)\downarrow N$ drops I rows and J columns from a matrix N, the vector D may be expressed as $\times/0\ 1\downarrow(¯1+\iota\rho\alpha)\phi\alpha\circ.-\alpha$; the definition of the function *STEP* may therefore be replaced by the more efficient definition:

```
STEP:((α∘.*¯1+⍳⍴ω)+.×ω)÷×/0 1↓(¯1+⍳⍴α)⌽α∘.-α          C.3
```

This last is the elegant method of Kerner [7]. Using starting values given by the left argument of G in C.2, it converges in seven steps (with a tolerance $TOL\leftarrow 1E^{-}8$) for the sixth-order example given by Kerner.

3.3 Permutations

A vector `P` whose elements are some permutation of its indices (that is, `∧/1=+/P∘.=⍳⍴P`) will be called a *permutation* vector. If `D` is a permutation vector such that `(⍴X)=⍴D`, then `X[D]` is a permutation of `X`, and `D` will be said to be the *direct* representation of this permutation.

The permutation `X[D]` may also be expressed as `B+.×X`, where `B` is the boolean matrix `D∘.=⍳⍴D`. The matrix `B` will be called the *boolean* representation of the permutation. The transformations between direct and boolean representations are:

```
BFD:ω∘.=⍳⍴ω          DFB:ω+.×⍳1+⍴ω
```

Because permutation is associative, the composition of permutations satisfies the following relations:

```
(X[D1])[D2] ↔ X[(D1 [D2])]
B2+.×(B1+.×X) ↔ (B2+.×B1)+.×X
```

The inverse of a boolean representation `B` is `⌹B`, and the inverse of a direct representation is either `⍋D` or `D⍳⍳⍴D`. (The *grade* function `⍋` grades its argument, giving a vector of indices to its elements in ascending order, maintaining existing order among equal elements. Thus `⍋3 7 1 4` is `3 1 4 2` and `⍋3 7 3 4` is `1 3 4 2`. The *index-of* function `⍳` determines the smallest index in its left argument of each element of its right argument. For example, `'ABCDE'⍳'BABE'` is `2 1 2 5`, and `'BABE'⍳'ABCDE'` is `2 1 5 5 4`.)

The *cycle* representation also employs a permutation vector. Consider a permutation vector `C` and the segments of `C` marked off by the vector `C=⍳\C`. For example, if `C←7 3 6 5 2 1 4`, then `C=⍳\C` is `1 1 0 0 1 1 0`, and the blocks are:

```
7
3 6 5
2
1 4
```

Each block determines a "cycle" in the associated permutation in the sense that if `R` is the result of permuting `X`, then:

```
R[7] is X[7]
R[3] is X[6]        R[6] is X[5]        R[5] is X[3]
R[2] is X[2]
R[1] is X[4]        R[4] is X[1]
```

If the leading element of `C` is the smallest (that is, `1`), then `C` consists of a single cycle, and the permutation of a vector `X` which it represents is given by `X[C]←X[1⌽C]`. For example:

```
      X←'ABCDEFG'
      C←1 7 6 5 2 4 3
      X[C]←X[1⌽C]
      X
GDACBEF
```

Since `X[Q]←A` is equivalent to `X←A[⍋Q]`, it follows that `X[C]←X[1⌽C]` is equivalent to `X←X[(1⌽C)[⍋C]]`, and the direct representation vector `D` equivalent to `C` is therefore given (for the special case of a single cycle) by `D←(1⌽C)[⍋C]`.

In the more general case, the rotation of the complete vector (that is, `1⌽C`) must be replaced by rotations of the individual subcycles marked off by `C=⍳\C`, as shown in the following definition of the transformation to direct from cycle representation:

```
DFC:(ω[⍋X++\X←ω=⍳\ω])[⍋ω]
```

If one wishes to catenate a collection of disjoint cycles to form a single vector `C` such that `C=⍳\C` marks off the individual cycles, then each cycle `CI` must first be brought to *standard form* by the rotation `(¯1+CI⍳⌊/CI)⌽CI`, and the resulting vectors must be catenated in descending order on their leading elements.

The inverse transformation from direct to cycle representation is more complex, but can be approached by first producing the matrix of all powers of `D` up to the `⍴D`th, that is, the matrix whose successive columns are `D` and `D[D]` and `(D[D])[D]`, etc. This is obtained by applying the function `POW` to the one-column matrix `D∘.+,0` formed from `D`, where `POW` is defined and used as follows:

```
      POW:POW D,(D←ω[;1])[ω]:≤/⍴ω:ω

      ⎕←D←DFC C←7,3 6 5,2,1 4
4 2 6 1 3 5 7
      POW D∘.+,0
 4 1 4 1 4 1 4
 2 2 2 2 2 2 2
 6 5 3 6 5 3 6
 1 4 1 4 1 4 1
 3 6 5 3 6 5 3
 5 3 6 5 3 6 5
 7 7 7 7 7 7 7
```

If `M←POW D∘.+,0`, then the cycle representation of `D` may be obtained by selecting from `M` only "standard" rows which begin with their smallest elements (`SSR`), by arranging these remaining rows in descending order on their leading elements (`DOL`), and then catenating the cycles in these rows (`CIR`). Thus:

```
      CFD:CIR DOL SSR POW ω∘.+,0

        SSR:(∧/M=1⌽M←⍳\ω)⌿ω
        DOL:ω[⍒ω[;1];]
        CIR:(,1,∧\0 1+ω≠⍳\ω)/,ω

      DFC C←7,3 6 5,2,1 4
4 2 6 1 3 5 7
      CFD DFC C
7 3 6 5 2 1 4
```

In the definition of `DOL`, indexing is applied to matrices. The indices for successive coordinates are separated by semicolons, and a blank entry for any axis indicates that all elements along it are selected. Thus `M[;1]` selects column `1` of `M`.

The cycle representation is convenient for determining the number of cycles in the permutation represented (`NC:+/ω=⍳\ω`), the cycle lengths (`CL:X-0,¯1+X←(1⌽ω=⍳\ω)/⍳⍴ω`), and the *power* of the permutation (`PP:LCM CL ω`). On the other hand, it is awkward for composition and inversion.

The `:N` column vectors of the matrix `(⌽⍳N)⍉¯1+⍳!N` are all distinct, and therefore provide

a potential *radix* representation [8] for the ⍳N permutations of order N. We will use instead a related form obtained by increasing each element by 1:

```
      RR:1+(⍵⍴⍳⍵)⊤⍳!⍵

      RR 4
1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4
1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Transformations between this representation and the direct form are given by:

```
      DFR:⍵[1],X+⍵[1]≤X←DFR 1+⍵:0=⍴⍵:⍵
      RFD:⍵[1],RFD X-⍵[1]≤X←1+⍵:0=⍴⍵:⍵
```

Some of the characteristics of this alternate representation are perhaps best displayed by modifying DFR to apply to all columns of a matrix argument, and applying the modified function MF to the result of the function RR:

```
      MF:⍵[,1;],[1]X+⍵[(1 ⍴X)⍴1;]≤X←MF 1 0+⍵:0=1+⍴⍵:⍵
      MF RR 4
1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4
2 2 3 3 4 4 1 1 3 3 4 4 1 1 2 2 4 4 1 1 2 2 3 3
3 4 2 4 2 3 3 4 1 4 1 3 2 4 1 4 1 2 2 3 1 3 1 2
4 3 4 2 3 2 4 3 4 1 3 1 4 2 4 1 2 1 3 2 3 1 2 1
```

The direct permutations in the columns of this result occur in *lexical* order (that is, in ascending order on the first element in which two vectors differ); this is true in general, and the alternate representation therefore provides a convenient way for producing direct representations in lexical order.

The alternate representation also has the useful property that the parity of the direct permutation D is given by 2|+/⍳⁻1+RFD D, where M|N represents the residue of N modulo M. The parity of a direct representation can also be determined by the function:

```
      PAR:2|+/,(I∘.>I←⍳⍴⍵)∧⍵∘.>⍵
```

3.4 Directed Graphs

A simple directed graph is defined by a set of K nodes and a set of directed connections from one to another of pairs of the nodes. The directed connections may be conveniently represented by a K by K boolean *connection* matrix C in which C[I;J]=1 denotes a connection *from* the Ith node *to* the Jth.

For example, if the four nodes of a graph are represented by N←'QRST', and if there are connections from node S to node Q, from R to T, and from T to Q, then the corresponding connection matrix is given by:

```
0 0 0 0
0 0 0 1
1 0 0 0
1 0 0 0
```

A connection from a node to itself (called a self-loop) is not permitted, and the diagonal of a connection matrix must therefore be zero.

If P is any permutation vector of order ⍴N, then N1←N[P] is a reordering of the nodes, and the corresponding connection matrix is given by C[P;P]. We may (and will) without loss of generality use the numeric labels ⍳⍴N for the nodes, because if N is any arbitrary vector of names for the nodes and L is any list of numeric labels, then the expression Q←N[L] gives the corresponding list of names and, conversely, N⍳Q gives the list L of numeric labels.

The connection matrix C is convenient for expressing many useful functions on a graph. For example, +/C gives the *out-degrees* of the nodes, +⌿C gives the *in-degrees*, +/,C gives the number of connections or *edges*, ⍉C gives a related graph with the directions of edges reversed, and C∨⍉C gives a related "symmetric" or "undirected" graph. Moreover, if we use the boolean vector B←∨/(⍳1⍴⍴C)∘.=L to represent the list of nodes L, then B∨.∧C gives the boolean vector which represents the set of nodes directly reachable from the set B. Consequently, C∨.∧C gives the connections for paths of length two in the graph C, and C∨C∨.∧C gives connections for paths of length one or two. This leads to the following function for the *transitive closure* of a graph, which gives all connections through paths of any length:

```
      TC:TC Z:∧/,⍵=Z←⍵∨⍵∨.∧⍵:Z
```

Node J is said to be *reachable* from node I if (TC C)[I;J]=1. A graph is *strongly-connected* if every node is reachable from every node, that is ∧/,TC C.

If D←TC C and D[I;I]=1 for some I, then node I is reachable from itself through a path of some length; the path is called a *circuit*, and node I is said to be contained in a circuit.

A graph T is called a *tree* if it has no circuits and its in-degrees do not exceed 1, that is, ∧/1≥+⌿T. Any node of a tree with an in-degree of 0 is called a *root*, and if K←+/0=+⌿T, then T is called a K-rooted tree. Since a tree is circuit-free, K must be at least 1. Unless otherwise stated, it is normally assumed that a tree is *singly-rooted* (that is, K=1); multiply-rooted trees are sometimes called *forests*.

A graph C *covers* a graph D if ∧/,C≥D. If G is a strongly-connected graph and T is a (singly-rooted) tree, then T is said to be a *spanning tree* of G if G covers T and if all nodes are reachable from the root of T, that is,

```
(∧/,G≥T) ∧ ∧/R∨R∨.∧TC T
```

where R is the (boolean representation of the) root of T.

A *depth-first spanning tree* [9] of a graph G is a spanning tree produced by proceeding from the root through immediate descendants in G, always choosing as the next node a descendant of the latest in the list of nodes visited which still possesses a descendant not in the list. This is a relatively

complex process which can be used to illustrate the utility of the connection matrix representation:

```
DFST:((,1)∘.=K) R ω∧K∘.∨~K←α=⍳1↑⍴ω                    C.4

R:(C,[1]α)Rω∧P∘.∨~C←<\U∧P∨.∧ω
  :~∨/P←(<\α∨.∧ω∨.∧U←~∨⌿α)∨.∧α
  :ω
```

Using as an example the graph G from [9]:

```
      G
0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1 1 0
0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0

      1 DFST G
0 0 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
```

The function $DFST$ establishes the left argument of the recursion R as the one-row matrix representing the root specified by the left argument of $DFST$, and the right argument as the original graph with the connections *into* the root K deleted. The first line of the recursion R shows that it continues by appending on the top of the list of nodes thus far assembled in the left argument the next child C, and by deleting from the right argument all connections into the chosen child C except the one from its parent P. The child C is chosen from among those reachable from the chosen parent ($P\vee.\wedge\omega$), but is limited to those as yet untouched ($U\wedge P\vee.\wedge\omega$), and is taken, arbitrarily, as the first of these ($<\backslash U\wedge P\vee.\wedge\omega$).

The determinations of P and U are shown in the second line, P being chosen from among those nodes which have children among the untouched nodes ($\omega\vee.\wedge U$). These are permuted to the order of the nodes in the left argument ($\alpha\vee.\wedge\omega\vee.\wedge U$), bringing them into an order so that the last visited appears first, and P is finally chosen as the first of these.

The last line of R shows the final result to be the resulting right argument ω, that is, the original graph with all connections into each node broken except for its parent in the spanning tree. Since the final value of α is a square matrix giving the nodes of the tree in reverse order as visited, substitution of $\omega,\phi[1]\alpha$ (or, equivalently, $\omega,\ominus\alpha$) for ω would yield a result of shape $1\ 2\times\rho G$ containing the spanning tree followed by its "preordering" information.

Another representation of directed graphs often used, at least implicitly, is the list of all node pairs V,W such that there is a connection from V to W. The transformation to this list form from the connection matrix may be defined and used as follows:

```
      LFC:(,ω)/1+⍳¯1+⍳×/D←⍴ω
    C                   LFC C
0 0 1 1             1 1 2 3 3 4
0 0 1 0             3 4 3 2 4 1
0 1 0 1
1 0 0 0
```

However, this representation is deficient since it does not alone determine the number of nodes in the graph, although in the present example this is given by $\lceil/,LFC\ C$ because the highest numbered node happens to have a connection. A related boolean representation is provided by the expression $(LFC\ C)\circ.=\iota 1\uparrow\rho C$, the first plane showing the out- and the second showing the in-connections.

An *incidence* matrix representation often used in the treatment of electric circuits [10] is given by the difference of these planes as follows:

```
IFC:-⌿(LFC ω)∘.=⍳1↑⍴ω
```

For example:

```
      (LFC C)∘.=⍳1↑⍴C
1 0 0 0
1 0 0 0
0 1 0 0
0 0 1 0
0 0 1 0
0 0 0 1

0 0 1 0
0 0 0 1
0 0 1 0
0 1 0 0
0 0 0 1
1 0 0 0

      IFC C
 1  0 ¯1  0
 1  0  0 ¯1
 0  1 ¯1  0
 0 ¯1  1  0
 0  0  1 ¯1
¯1  0  0  1
```

In dealing with non-directed graphs, one sometimes uses a representation derived as the *or* over these planes ($\vee\not$). This is equivalent to $\lvert IFC\ C$.

The incidence matrix I has a number of useful properties. For example, $+/I$ is zero, $+\not I$ gives the *difference* between the in- and out-degrees of each node, ρI gives the number of edges followed by the number of nodes, and $\times/\rho I$ gives their product. However, all of these are also easily expressed in terms of the connection matrix, and more significant properties of the incidence matrix are seen in its use in electric circuits. For example, if the edges represent components connected between the nodes, and if V is the vector of node voltages, then the branch voltages are given by $I+.\times V$; if BI is the vector of branch currents, the vector of node currents is given by $BI+.\times I$.

The inverse transformation from incidence matrix to connection matrix is given by:

```
CFI:D⍴(¯1+⍳×/D)∈D⊥(1 ¯1∘.=ω)+.×¯1+⍳1↑⍴D←⌊\⌽⍴ω
```

The *set membership* function $\in$ yields a boolean array, of the same shape as its left argument, which shows which of its elements belong to the right argument.

3.5 Symbolic Logic

A boolean function of N arguments may be represented by a boolean vector of $2*N$ elements in a variety of ways, including what are sometimes called the *disjunctive, conjunctive, equivalence,* and *exclusive-disjunctive* forms. The transformation between any pair of these forms may be represented concisely as some $2*N$ by $2*N$ matrix formed

by a related inner product, such as `T∨.∧⍉T`, where `T ← T N` is the "truth table" formed by the function `T` defined by A.2. These matters are treated fully in [11, Ch.7].

4. Identities and Proofs

In this section we will introduce some widely used identities and provide formal proofs for some of them, including Newton's symmetric functions and the associativity of inner product, which are seldom proved formally.

4.1 Dualities in Inner Products

The dualities developed for reduction and scan extend to inner products in an obvious way. If `DF` is the dual of `F` and `DG` is the dual of `G` with respect to a monadic function `M` with inverse `MI`, and if `A` and `B` are matrices, then:

```
A F.G B ↔ MI (M A) DF.DG (M B)
```

For example:

```
A∨.∧B ↔ ~(~A)∧.∨(~B)
A∧.=B ↔ ~(~A)∨.≠(~B)
A⌊.+B ↔ -(-A)⌈.+(-B)
```

The dualities for inner product, reduction, and scan can be used to eliminate many uses of boolean negation from expressions, particularly when used in conjunction with identities of the following form:

```
   A∧(~B) ↔ A>B
(~A)∧B    ↔ A<B
(~A)∧(~B) ↔ A≠B
```

4.2 Partitioning Identities

Partitioning of an array leads to a number of obvious and useful identities. For example:

```
×/3 1 4 2 6 ↔ (×/3 1) × (×/4 2 6)
```

More generally, for any associative function `F`:

```
  F/V ↔ (F/K↑V) F (F/K↓V)
F/V,W ↔ (F/V) F (F/W)
```

If `F` is commutative as well as associative, the partitioning need not be limited to prefixes and suffixes, and the partitioning can be made by compression by a boolean vector `U`:

```
F/V ↔ (F/U/V) F (F/(~U)/V)
```

If `E` is an empty vector `(0=⍴E)`, the reduction `F/E` yields the identity element of the function `F`, and the identities therefore hold in the limiting cases `0=K` and `0=∨/U`.

Partitioning identities extend to matrices in an obvious way. For example, if `V`, `M`, and `A` are arrays of ranks `1`, `2`, and `3`, respectively, then:

```
V+.×M ↔ ((K↑V)+.×(K,1↓⍴M)↑M)+(K↓V)+.×(K,0)↑M      D.1
(I,J)↑A+.×V ↔ ((I,J,0)↑A)+.×V                     D.2
```

4.3 Summarization and Distribution

Consider the definition and and use of the following functions:

```
N:(∨⌿<\ω∘.=ω)/ω                                   D.3
S:(Nω)∘.=ω                                        D.4
```

```
      A←3 3 1 4 1
      C←10 20 30 40 50

      N A          S A              (S A)+.×C
3 1 4        1 1 0 0 0        30 80 40
             0 0 1 0 1
             0 0 0 1 0
```

The function `N` selects from a vector argument its *nub*, that is, the set of distinct elements it contains. The expression `S A` gives a boolean "summarization matrix" which relates the elements of `A` to the elements of its nub. If `A` is a vector of account numbers and `C` is an associated vector of costs, then the expression `(S A)+.×C` evaluated above sums or "summarizes" the charges to the several account numbers occurring in `A`.

Used as postmultiplier, in expressions of the form `W+.×S A`, the summarization matrix can be used to *distribute* results. For example, if `F` is a function which is costly to evaluate and its argument `V` has repeated elements, it may be more efficient to apply `F` only to the nub of `V` and distribute the results in the manner suggested by the following identity:

```
F V ↔ (F N V)+.×S V                               D.5
```

The order of the elements of `N V` is the same as their order in `V`, and it is sometimes more convenient to use an *ordered* nub and corresponding *ordered* summarization given by:

```
QN:Nω[⍋ω]                                         D.6
QS:(QNω)∘.=ω                                      D.7
```

The identity corresponding to D.5 is:

```
F V ↔ (F QN V)+.×QS V                             D.8
```

The summarization function produces an interesting result when applied to the function `T` defined by A.2:

```
+/S+⌿T N ↔ (0,⍳N)!N
```

In words, the sums of the rows of the summarization matrix of the column sums of the subset matrix of order `N` is the vector of binomial coefficients of order `N`.

4.4 Distributivity

The distributivity of one function over another is an important notion in mathematics, and we will now raise the question of representing this in a general way. Since multiplication distributes to the right over addition we have `a×(b+q)↔ab+aq` , and since it distributes to the left we have `(a+p)×b↔ab+pb`. These lead to the more general cases:

```
(a+p)×(b+q) ↔ ab+aq+pb+pq
(a+p)×(b+q)×(c+r) ↔ abc+abr+aqc+aqr+pbc+pbr+pqc+pqr
(a+p)×(b+q)×...×(c+r) ↔ ab...c+....+pq...r
```

Using the notion that `V←A,B` and `W←P,Q` or `V←A,B,C` and `W←P,Q,R`, etc., the left side can be written simply in terms of reduction as `×/V+W`. For this case of three elements, the right side can be written as the sum of the products over the columns of the following matrix:

```
V[0]  V[0]  V[0]  V[0]  W[0]  W[0]  W[0]  W[0]
V[1]  V[1]  W[1]  W[1]  V[1]  V[1]  W[1]  W[1]
V[2]  W[2]  V[2]  W[2]  V[2]  W[2]  V[2]  W[2]
```

The pattern of `V`'s and `W`'s above is precisely the pattern of *zeros* and *ones* in the matrix `T←⍉⍳2*⍴V`, and so the products down the columns are given by `(V×.*~T)×(W×.*T)`. Consequently:

```
×/V+W ↔ +/(V×.*~T)×W×.*T←⍉⍳2*⍴V                    D.9
```

We will now present a formal inductive proof of D.9, assuming as the induction hypothesis that D.9 is true for all `V` and `W` of shape `N` (that is, `∧/N=(⍴V),⍴W`) and proving that it holds for shape `N+1`, that is, for `X,V` and `Y,W`, where `X` and `Y` are arbitrary scalars.

For use in the inductive proof we will first give a recursive definition of the function `⍉`, equivalent to A.2 and based on the following notion: if `M←⍉2` is the result of order 2, then:

```
      M
0 0 1 1
0 1 0 1
      0,[1]M          1,[1]M
0 0 0 0         1 1 1 1
0 0 1 1         0 0 1 1
0 1 0 1         0 1 0 1

      (0,[1]M),(1,(1)M)
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
```

Thus:

```
      ⍉:(0,[1]⍉),(1,[1]⍉←⍉ω-1):0=ω:0 1⍴0                    D.10

+/((C←X,V)×.*~Q)×D×.*Q←⍉⍴(D←Y,W)
+/(C×.*~Z,U)×D×.*(Z←0,[1]⍉),U←1,[1]⍉←⍉⍴W                    D.10
+/((C×.*~Z),C×.*~U)×(D×.*Z),D×.*U                          Note 1
+/((C×.*~Z),C×.*~U)×((Y*0)×W×.*⍉),(Y*1)×W×.*⍉              Note 2
+/((C×.*~Z),C×.*~U)×(W×.*⍉),Y×W×.*⍉          Y*0 1↔1,Y
+/((X×V×.*~⍉),V×.*~⍉)×(W×.*⍉),Y×W×.*⍉                      Note 2
+/(X×(V×.*~⍉)×W×.*⍉),(Y×(V×.*~⍉)×W×.*⍉)                    Note 3
+/(X××/V+W),(Y××/V+W)                          Induction hypothesis
+/(X,Y)××/V+W                                (X×S),(Y×S)↔(X,Y)×S
×/(X+Y),(V+W)                                  Definition of ×/
×/(X,V)+(Y,W)                                  + distributes over ,
```

Note 1: `M+.×N,P ↔ (M+.×N),M+.×P` (partitioning identity on matrices)

Note 2: `V+.×M ↔ ((1+V)+.×(1,1+⍴M)↑M)+(1+V)+.×1 0↓M` (partitioning identity on matrices and the definition of `C`, `D`, `Z`, and `U`)

Note 3: `(V,W)×P,Q ↔ (V×P),W×Q`

To complete the inductive proof we must show that the putative identity D.9 holds for some value of `N`. If `N=0`, the vectors `A` and `B` are empty, and therefore `X,A ↔ ,X` and `Y,B ↔ ,Y`. Hence the left side becomes `×/X+Y`, or simply `X+Y`. The right side becomes `+/(X×.*~Q)×Y×.*Q`, where `~Q` is the one-rowed matrix `1 0` and `Q` is `0 1`. The right side is therefore equivalent to `+/(X,1)×(1,Y)`, or `X+Y`. Similar examination of the case `N=1` may be found instructive.

4.5 Newton's Symmetric Functions

If `X` is a scalar and `R` is any vector, then `×/X-R` is a polynomial in `X` having the roots `R`. It is therefore equivalent to some polynomial `C ⍎ X`, and assumption of this equivalence implies that `C` is a function of `R`. We will now use D.8 and D.9 to derive this function, which is commonly based on Newton's symmetric functions:

```
×/X-R
×/X+(-R)
+/(X×.*~T)×(-R)×.*T←⍉⍴R                                    D.9
(X×.*~T)+.×P←(-R)×.*T                                  Def of +.×
(X*S←+⌿~T)+.×P                                         Note 1
((X*QN S)+.×QS S)+.×P                                  D.8
(X*QN S)+.×((QS S)+.×P)                          +.× is associative
(X*0,⍳⍴R)+.×((QS S)+.×P)                               Note 2
((QS S)+.×P)⍎X                                   B.1 (polynomial)
((QS +⌿~T)+.×((-R)×.*T←⍉⍴R))⍎X                         Defs of S
                                                          and P
```

Note 1: If `X` is a *scalar* and `B` is a boolean vector, then `X×.*B ↔ X*+/B`.

Note 2: Since `T` is boolean and has `⍴R` rows, the sums of its columns range from 0 to `⍴R`, and their ordered nub is therefore `0,⍳⍴R`.

4.6 Dyadic Transpose

The dyadic transpose, denoted by `⍉`, is a generalization of monadic transpose which permutes axes of the right argument, and (or) forms "sectors" of the right argument by coalescing certain axes, all as determined by the left argument. We introduce it here as a convenient tool for treating properties of the inner product.

The dyadic transpose will be defined formally in terms of the selection function

```
SF:(⍴ω)[1+(⍴ω)⊥α-1]
```

which selects from its right argument the element whose indices are given by its vector left argument, the shape of which must clearly equal the rank of the right argument. The rank of the result of `K⍉A` is `⌈/K`, and if `I` is any suitable left argument of the selection `I SF K⍉A` then:

```
I SF K⍉A ↔ (I[K]) SF A                                     D.11
```

For example, if `M` is a matrix, then `2 1 ⍉M ↔ ⍉M` and `1 1 ⍉M` is the diagonal of `M`; if `T` is a rank three array, then `1 2 2 ⍉T` is a matrix "diagonal section" of `T` produced by running together the last two axes, and the vector `1 1 1 ⍉T` is the principal body diagonal of `T`.

The following identity will be used in the sequel:

```
J⍉K⍉A ↔ (J[K])⍉A                                          D.12
```

Proof:

```
I SF J⍉K⍉A
(I[J]) SF K⍉A                                  Definition of ⍉ (D.11)
((I[J])[K]) SF A                               Definition of ⍉
(I[(J[K])]) SF A                               Indexing is associative
I SF(J[K])⍉A                                   Definition of ⍉
```

4.7 Inner Products

The following proofs are stated only for matrix arguments and for the particular inner product `+.×`. They are easily extended to arrays of higher rank and to other inner products `F.G`, where `F` and `G` need possess only the properties assumed in the proofs for `+` and `×`.

The following identity (familiar in mathematics as a sum over the matrices formed by (outer) products of columns of the first argument with corresponding rows of the second argument) will be used in establishing the associativity and distributivity of the inner product:

```
M+.×N ↔ +/1 3 3 2 ⍉ M∘.×N                                    D.13
```

Proof: `(I,J)SF M+.×N` is defined as the sum over `V`, where `V[K] ↔ M[I;K]×N[K;J]`. Similarly,

```
(I,J)SF +/1 3 3 2 ⍉ M∘.×N
```

is the sum over the vector `W` such that

```
W[K] ↔ (I,J,K)SF 1 3 3 2 ⍉ M∘.×N
```

Thus:

```
W[K]
(I,J,K)SF 1 3 3 2 ⍉M∘.×N
(I,J,K)[1 3 3 2]SF M∘.×N                                      D.12
(I,K,K,J)SF M∘.×N                                  Def of indexing
M[I;K]×N[K;J]                                  Def of Outer product
V[K]
```

Matrix product distributes over addition as follows:

```
M+.×(N+P) ↔ (M+.×N)+(M+.×P)                                   D.14
```

Proof:

```
M+.×(N+P)
+/(J← 1 3 3 2)⍉M∘.×N+P                                        D.13
+/J⍉(M∘.×N)+(M∘.×P)                          × distributes over +
+/(J⍉M∘.×N)+(J⍉M∘.×P)                        ⍉ distributes over +
(+/J⍉M∘.×N)+(+/J⍉M∘.×P)                      + is assoc and comm
(M+.×N)+(M+.×P)                                               D.13
```

Matrix product is associative as follows:

```
M+.×(N+.×P) ↔ (M+.×N)+.×P                                     D.15
```

Proof: We first reduce each of the sides to sums over sections of an outer product, and then compare the sums. Annotation of the second reduction is left to the reader:

```
M+.×(N+.×P)
M+.×+/1 3 3 2⍉N∘.×P                                           D.12
+/1 3 3 2⍉M∘.×+/1 3 3 2⍉N∘.×P                                 D.12
+/1 3 3 2⍉+/M∘.×1 3 3 2⍉N∘.×P                × distributes over +
+/1 3 3 2⍉+/1 2 3 5 5 4⍉M∘.×N∘.×P                            Note 1
+/+/1 3 3 2 4 ⍉1 2 3 5 5 4⍉M∘.×N∘.×P                         Note 2
+/+/1 3 3 4 4 2⍉M∘.×N∘.×P                                     D.12
+/+/1 3 3 4 4 2⍉(M∘.×N)∘.×P                   × is associative
+/+/1 4 4 3 3 2⍉(M∘.×N)∘.×P                   + is associative and
                                                              commutative

(M+.×N)+.×P
(+/1 3 3 2⍉M∘.×N)+.×P
+/1 3 3 2⍉(+/1 3 3 2⍉M∘.×N)∘.×P
+/1 3 3 2⍉+/1 5 5 2 3 4⍉(M∘.×N)∘.×P
+/+/1 3 3 2 4⍉1 5 5 2 3 4⍉(M∘.×N)∘.×P
+/+/1 4 4 3 3 2⍉(M∘.×N)∘.×P
```

Note 1: `+/M∘.×J⍉A ↔ +/((⍳⍴⍴M),J+⍴⍴M)⍉M∘.×A`

Note 2: `J⍉+/A ↔ +/(J,1+⌈/J)⍉A`

4.8 Product of Polynomials

The identity B.2 used for the multiplication of polynomials will now be developed formally:

```
(B ⍒ X)×(C ⍒ X)
(+/B×X*E←¯1+⍳⍴B)×(+/C×X*F←¯1+⍳⍴C)                             B.1
+/+/(B×X*E)∘.×(C×X*F)                                        Note 1
+/+/(B∘.×C)×((X*E)∘.×(X*F))                                  Note 2
+/+/(B∘.×C)×(X*(E∘.+F))                                      Note 3
```

Note 1: `(+/V)×(+/W)↔+/+/V∘.×X` because × distributes over +and + is associative and commutative, or see [12,P21] for a proof.

Note 2: The equivalence of `(P×V)∘.×(Q×W)` and `(P∘.×Q)×(V∘.×W)` can be established by examining a typical element of each expression.

Note 3: `(X*I)×(X*J)↔X*(I+J)`

The foregoing is the proof presented, in abbreviated form, by Orth [13, p.52], who also defines functions for the composition of polynomials.

4.9 Derivative of a Polynomial

Because of their ability to approximate a host of useful functions, and because they are closed under addition, multiplication, composition, differentiation, and integration, polynomial functions are very attractive for use in introducing the study of calculus. Their treatment in elementary calculus is, however, normally delayed because the derivative of a polynomial is approached indirectly, as indicated in Section 2, through a sequence of more general results.

The following presents a derivation of the derivative of a polynomial directly from the expression for the slope of the secant line through the points `X`, `F X` and `(X+Y),F(X+Y)`:

```
((C ⍒ X+Y)-(C ⍒ X))÷Y
((C ⍒ X+Y)-(C ⍒ X+0))÷Y
((C ⍒ X+Y)-((0*J)+.×(A←DS J∘.!J←¯1+⍳⍴C)+.×C) ⍒ X)÷Y            B.6
((((Y*J)+.×M) ⍒ X)-((0*J)+.×M←A+.×C) ⍒ X)÷Y                    B.6
((((Y*J)+.×M)-(0*J)+.×M) ⍒ X)÷Y                    ⍒ dist over -
((((Y*J)-0*J)+.×M) ⍒ X)÷Y                          +.× dist over -
(((0,Y*1+J)+.×M) ⍒ X)÷Y                                      Note 1
(((Y*1+J)+.× 1 0 +M) ⍒ X)÷Y                                  D.1
(((Y*1+J)+.×(1 0 0 +A)+.×C) ⍒ X)÷Y                           D.2
((Y*1+J-1)+.×(1 0 0 +A)+.×C) ⍒ X                  (Y*A)÷Y↔Y*A-1
((Y*¯1+⍳¯1+⍴C)+.×(1 0 0 +A)+.×C) ⍒ X                        Def of J
(((Y*¯1+⍳¯1+⍴C)+.× 1 0 0 +A)+.×C) ⍒ X                        D.15
```

Note 1: `0*0↔1↔Y*0` and `∧/0=0*1+J`

The derivative is the limiting value of the secant slope for `Y` at zero, and the last expression above can be evaluated for this case because if `E←¯1+⍳¯1+⍴C` is the vector of exponents of `Y`, then all elements of `E` are non-negative. Moreover, `0*E` reduces to a `1` followed by zeros, and the inner product with `1 0 0+A` therefore reduces to the first plane of `1 0 0+A` or, equivalently, the second plane of `A`.

If `B←J∘.!J←¯1+⍳⍴C` is the matrix of binomial coefficients, then `A` is `DS B` and, from the definition of `DS` in B.5, the second plane of `A` is `B×1=-J∘.-J`, that is, the matrix `B` with all but the first super-diagonal replaced by zeros. The final expression for the coefficients of the polynomial which is the derivative of the polynomial `C ⍒ ω` is therefore:

```
((J∘.!J)×1=-J∘.-J←¯1+⍳⍴C)+.×C
```

For example:

```
      C ← 5 7 11 13
      (J∘.!J)×1=-J∘.-J←¯1+⍳⍴C
 0 1 0 0
 0 0 2 0
 0 0 0 3
 0 0 0 0
      ((J∘.!J)×1=-J∘-J←¯1+⍳⍴C)+.×C
7 22 39 0
```

Since the superdiagonal of the binomial coefficient matrix `(⍳N)∘.!⍳N` is `(¯1+⍳N-1)!⍳N-1`, or simply `⍳N-1`, the final result is `1⌽C×¯1+⍳⍴C` in agreement with the earlier derivation.

In concluding the discussion of proofs, we will re-emphasize the fact that all of the statements in the foregoing proofs are executable, and that a computer can therefore be used to identify errors. For example, using the canonical function definition mode [4 , p.81], one could define a function `F` whose statements are the first four statements of the preceding proof as follows:

```
   ∇F
[1] ((C ⍒ X+Y)-(C ⍒ X))÷Y
[2] ((C ⍒ X+Y)-(C ⍒ X+0))÷Y
[3] ((C ⍒ X+Y)-((0*J)+.×(A←DS J∘.!J←¯1+⍳⍴C)+.×C) ⍒ X)÷Y
[4] ((((Y*J)+.×M) ⍒ X)-((0*J)+.×M←A+.×C) ⍒ X)÷Y
   ∇
```

The statements of the proof may then be executed by assigning values to the variables and executing `F` as follows:

```
      C←5 2 3 1
      Y←5
      X←3          X←⍳10
      F            F
132       66 96 132 174 222 276 336 402 474 552
132       66 96 132 174 222 276 336 402 474 552
132       66 96 132 174 222 276 336 402 474 552
132       66 96 132 174 222 276 336 402 474 552
```

The annotations may also be added as comments between the lines without affecting the execution.

5. Conclusion

The preceding sections have attempted to develop the thesis that the properties of executability and universality associated with programming languages can be combined, in a single language, with the well-known properties of mathematical notation which make it such an effective tool of thought. This is an important question which should receive further attention, regardless of the success or failure of this attempt to develop it in terms of APL.

In particular, I would hope that others would treat the same question using other programming languages and conventional mathematical notation. If these treatments addressed a common set of topics, such as those addressed here, some objective comparisons of languages could be made. Treatments of some of the topics covered here are already available for comparison. For example, Kerner [7] expresses the algorithm C.3 in both ALGOL and conventional mathematical notation.

This concluding section is more general, concerning comparisons with mathematical notation, the problems of introducing notation, extensions to APL which would further enhance its utility, and discussion of the mode of presentation of the earlier sections.

5.1 Comparison with Conventional Mathematical Notation

Any deficiency remarked in mathematical notation can probably be countered by an example of its rectification in some particular branch of mathematics or in some particular publication; comparisons made here are meant to refer to the more general and commonplace use of mathematical notation.

APL is similar to conventional mathematical notation in many important respects: in the use of functions with explicit arguments and explicit results, in the concomitant use of composite expressions which apply functions to the results of other functions, in the provision of graphic symbols for the more commonly used functions, in the use of vectors, matrices, and higher-rank arrays, and in the use of operators which, like the derivative and the convolution operators of mathematics, apply to functions to produce functions.

In the treatment of functions APL differs in providing a precise formal mechanism for the definition of new functions. The direct definition form used in this paper is perhaps most appropriate for purposes of exposition and analysis, but the canonical form referred to in the introduction, and defined in [4, p.81], is often more convenient for other purposes.

In the interpretation of composite expressions APL agrees in the use of parentheses, but differs in eschewing hierarchy so as to treat all functions (user-defined as well as primitive) alike, and in adopting a single rule for the application of both monadic and dyadic functions: the right argument of a function is the value of the entire expression to its right. An important consequence of this rule is that any portion of an expression which is free of parentheses may be read *analytically* from left to right (since the leading function at any stage is the "outer" or overall function to be applied to the result on its right), and *constructively* from right to left (since the rule is easily seen to be equivalent to the rule that *execution* is carried out from right to left).

Although Cajori does not even mention rules for the order of execution in his two-volume history of mathematical notations, it seems reasonable to assume that the motivation for the familiar hierarchy (power before × and × before + or -) arose from a desire to make polynomials expressible without parentheses. The convenient use of vec-

Fig. 3.

$$\sum_{j=1}^{n} j \cdot 2^{-j}$$

$$1 \cdot 2 \cdot 3 + 2 \cdot 3 \cdot 4 + \ldots n \text{ terms} \longleftrightarrow \frac{1}{4} n(n+1)(n+2)(n+3)$$

$$1 \cdot 2 \cdot 3 \cdot 4 + 2 \cdot 3 \cdot 4 \cdot 5 + \ldots n \text{ terms} \longleftrightarrow \frac{1}{5} n(n+1)(n+2)(n+3)(n+4)$$

$$\frac{\left[\frac{x-a}{N}\right]^{-q}}{\Gamma(-q)} \sum_{j=0}^{N-1} \frac{\Gamma(j-q)}{\Gamma(j+1)} f\left(x - j\left[\frac{x-a}{N}\right]\right)$$

tors in expressing polynomials, as in `+/C×X*E`, does much to remove this motivation. Moreover, the rule adopted in APL also makes Horner's efficient expression for a polynomial expressible without parentheses:

```
+/3 4 2 5×X*0 1 2 3 ↔ 3+X×4+X×2+X×5
```

In providing graphic symbols for commonly used functions APL goes much farther, and provides symbols for functions (such as the power function) which are implicitly denied symbols in mathematics. This becomes important when operators are introduced; in the preceding sections the inner product `×.*` (which must employ a symbol for power) played an equal role with the ordinary inner product `+.×`. Prohibition of elision of function symbols (such as `×`) makes possible the unambigious use of multi-character names for variables and functions.

In the use of arrays APL is similar to mathematical notation, but more systematic. For example, `V+W` has the same meaning in both, and in APL the definitions for other functions are extended in the same element-by-element manner. In mathematics, however, expressions such as `V×W` and `V*W` are defined differently or not at all.

For example, `V×W` commonly denotes the *vector product* [14, p.308]. It can be expressed in various ways in APL. The definition

```
VP:((1φα)×¯1φω)-(¯1φα)×1φω
```

provides a convenient basis for an obvious proof that `VP` is "anticommutative" (that is, `V VP W ↔ -W VP V`), and (using the fact that `¯1φX ↔ 2φX` for 3-element vectors) for a simple proof that in 3-space `V` and `W` are both orthogonal to their vector product, that is, `∧/0=V+.×V VP W` and `∧/0=W+.×V VP W`.

APL is also more systematic in the use of operators to produce functions on arrays: reduction provides the equivalent of the sigma and pi notation (in `+/` and `×/`) and a host of similar useful cases; outer product extends the outer product of tensor anaysis to functions other than `×`, and inner product extends ordinary matrix product (`+.×`) to many cases, such as `∨.∧` and `⌊.+`, for which ad hoc definitions are often made.

The similarities between APL and conventional notation become more apparent when one learns a few rather mechanical substitutions, and the translation of mathematical expressions is instructive. For example, in an expression such as the first shown in Figure 3, one simply substitutes `⍳N` for each occurrence of j and replaces the sigma by `+/`. Thus:

```
+/(⍳N)×2*-⍳N , or +/J×2*-J←⍳N
```

Collections such as Jolley's *Summation of Series* [15] provide interesting expressions for such an exercise, particularly if a computer is available for execution of the results. For example, on pages 8 and 9 we have the identities shown in the second and third examples of Figure 3. These would be written as:

```
+/×/(¯1+⍳N)∘.+⍳3 ↔ (×/N+0,⍳3)÷4
+/×/(¯1+⍳N)∘.+⍳4 ↔ (×/N+0,⍳4)÷5
```

Together these suggest the following identity:

```
+/×/(¯1+⍳N)∘.+⍳K ↔ (×/N+0,⍳K)÷K+1
```

The reader might attempt to restate this general identity (or even the special case where `K=0`) in Jolley's notation.

The last expression of Figure 3 is taken from a treatment of the fractional calculus [16, p.30], and represents an approximation to the qth order derivative of a function f. It would be written as:

```
(S*-Q)×+/(J!J-1+Q)×F X-(J←¯1+⍳N)×S←(X-A)÷N
```

The translation to APL is a simple use of `⍳N` as suggested above, combined with a straightforward identity which collapses the several occurrences of the gamma function into a single use of the binomial coefficient function `!`, whose domain is, of course, not restricted to integers.

In the foregoing, the parameter `Q` specifies the order of the derivative if positive, and the order of

the integral (from A to X) if negative. Fractional values give fractional derivatives and integrals, and the following function can, by first defining a function F and assigning suitable values to N and A, be used to experiment numerically with the derivatives discussed in [16]:

```
OS:(S*-⍺)×+/(J!J-1+⍺)×Fω-(J←¯1+⍳N)×S←(ω-A)÷N
```

Although much use is made of "formal" manipulation in mathematical notation, truly formal manipulation by explicit algorithms is very difficult. APL is much more tractable in this respect. In Section 2 we saw, for example, that the derivative of the polynomial expression $(\omega\circ.*{}^{-}1+\iota\rho\alpha)+.\times\alpha$ is given by $(\omega\circ.*{}^{-}1+\iota\rho\alpha)+.\times 1\phi\alpha\times{}^{-}1+\iota\rho\alpha$, and a set of functions for the formal differentiation of APL expressions given by Orth in his treatment of the calculus [13] occupies less than a page. Other examples of functions for formal manipulation occur in [17, p.347] in the modeling operators for the vector calculus.

Further discussion of the relationship with mathematical notation may be found in [3] and in the paper "Algebra as a Language" [6, p.325].

A final comment on printing, which has always been a serious problem in conventional notation. Although APL does employ certain symbols not yet generally available to publishers, it employs only 88 basic characters, plus some composite characters formed by superposition of pairs of basic characters. Moreover, it makes no demands such as the inferior and superior lines and smaller type fonts used in subscripts and superscripts.

5.2 The Introduction of Notation

At the outset, the ease of introducing notation in context was suggested as a measure of suitability of the notation, and the reader was asked to observe the process of introducing APL. The utility of this measure may well be accepted as a truism, but it is one which requires some clarification.

For one thing, an ad hoc notation which provided exactly the functions needed for some particular topic would be easy to introduce in context. It is necessary to ask further questions concerning the total bulk of notation required, the degree of structure in the notation, and the degree to which notation introduced for a specific purpose proves more generally useful.

Secondly, it is important to distinguish the difficulty of describing and of learning a piece of notation from the difficulty of mastering its implications. For example, learning the rules for computing a matrix product is easy, but a mastery of its implications (such as its associativity, its distributivity over addition, and its ability to represent linear functions and geometric operations) is a different and much more difficult matter.

Indeed, the very suggestiveness of a notation may make it seem harder to learn because of the many properties it suggests for exploration. For example, the notation $+.\times$ for matrix product cannot make the rules for its computation more difficult to learn, since it at least serves as a reminder that the process is an addition of products, but any discussion of the properties of matrix product in terms of this notation cannot help but suggest a host of questions such as: Is $\vee.\wedge$ associative? Over what does it distribute? Is $B\vee.\wedge C\leftrightarrow\sim(\sim C)\vee.\wedge\sim B$ a valid identity?

5.3 Extensions to APL

In order to ensure that the notation used in this paper is well-defined and widely available on existing computer systems, it has been restricted to current APL as defined in [4] and in the more formal standard published by STAPL, the ACM SIGPLAN Technical Committee on APL [17, p.409]. We will now comment briefly on potential extensions which would increase its convenience for the topics treated here, and enhance its suitability for the treatment of other topics such as ordinary and vector calculus.

One type of extension has already been suggested by showing the execution of an example (roots of a polynomial) on an APL system based on complex numbers. This implies no change in function symbols, although the domain of certain functions will have to be extended. For example, $|X$ will give the magnitude of complex as well as real arguments, $+X$ will give the conjugate of complex arguments as well as the trivial result it now gives for real arguments, and the elementary functions will be appropriately extended, as suggested by the use of $*$ in the cited example. It also implies the possibility of meaningful inclusion of primitive functions for zeros of polynomials and for eigenvalues and eigenvectors of matrices.

A second type also suggested by the earlier sections includes functions defined for particular purposes which show promise of general utility. Examples include the *nub* function $\subset$, defined by D.3, and the *summarization* function $\subseteq$, defined by D.4. These and other extensions are discussed in [18]. McDonnell [19, p.240] has proposed generalizations of *and* and *or* to non-booleans so that $A\vee B$ is the GCD of A and B, and $A\wedge B$ is the LCM. The functions GCD and LCM defined in Section 3 could then be defined simply by $GCD:\vee/\omega$ and $LCM:\wedge/\omega$.

A more general line of development concerns operators, illustrated in the preceding sections by the reduction, inner-product, and outer-product. Discussions of operators now in APL may be found

in [20] and in [17, p.129], proposed new operators for the vector calculus are discussed in [17, p.47], and others are discussed in [18] and in [17, p.129].

5.4 Mode of Presentation

The treatment in the preceding sections concerned a set of brief topics, with an emphasis on clarity rather than efficiency in the resulting algorithms. Both of these points merit further comment.

The treatment of some more complete topic, of an extent sufficient for, say, a one- or two-term course, provides a somewhat different, and perhaps more realistic, test of a notation. In particular, it provides a better measure of the amount of notation to be introduced in normal course work.

Such treatments of a number of topics in APL are available, including: high school algebra [6], elementary analysis [5], calculus, [13], design of digital systems [21], resistive circuits [10], and crystallography [22]. All of these provide indications of the ease of introducing the notation needed, and one provides comments on experience in its use. Professor Blaauw, in discussing the design of digital systems [21], says that "APL makes it possible to describe what really occurs in a complex system", that "APL is particularly suited to this purpose, since it allows expression at the high architectural level, at the lowest implementation level, and at all levels between", and that "....learning the language pays of (sic) in- and outside the field of computer design".

Users of computers and programming languages are often concerned primarily with the efficiency of execution of algorithms, and might, therefore, summarily dismiss many of the algorithms presented here. Such dismissal would be short-sighted, since a clear statement of an algorithm can usually be used as a basis from which one may easily derive more efficient algorithms. For example, in the function *STEP* of section 3.2, one may significantly increase efficiency by making substitutions of the form $B\boxplus M$ for $(\boxplus M)+.\times B$, and in expressions using $+/C\times X\star{}^{-}1+\iota\rho C$ one may substitute $X\perp\phi C$ or, adopting an opposite convention for the order of the coefficients, the expression $X\perp C$.

More complex transformations may also be made. For example, Kerner's method (C.3) results from a rather obvious, though not formally stated, identity. Similarly, the use of the matrix α to represent permutations in the recursive function R used in obtaining the depth first spanning tree (C.4) can be replaced by the possibly more compact use of a list of nodes, substituting indexing for inner products in a rather obvious, though not completely formal, way. Moreover, such a recursive definition can be transformed into more efficient non-recursive forms.

Finally, any algorithm expressed clearly in terms of arrays can be transformed by simple, though tedious, modifications into perhaps more efficient algorithms employing iteration on scalar elements. For example, the evaluation of $+/X$ depends upon every element of X and does not admit of much improvement, but evaluation of $\vee/B$ could stop at the first element equal to 1, and might therefore be improved by an iterative algorithm expressed in terms of indexing.

The practice of first developing a clear and precise definition of a process without regard to efficiency, and then using it as a guide and a test in exploring equivalent processes possessing other characteristics, such as greater efficiency, is very common in mathematics. It is a very fruitful practice which should not be blighted by premature emphasis on efficiency in computer execution.

Measures of efficiency are often unrealistic because they concern counts of "substantive" functions such as multiplication and addition, and ignore the housekeeping (indexing and other selection processes) which is often greatly increased by less straightforward algorithms. Moreover, realistic measures depend strongly on the current design of computers and of language embodiments. For example, because functions on booleans (such as $\wedge/B$ and $\vee/B$) are found to be heavily used in APL, implementers have provided efficient execution of them. Finally, overemphasis of efficiency leads to an unfortunate circularity in design: for reasons of efficiency early programming languages reflected the characteristics of the early computers, and each generation of computers reflects the needs of the programming languages of the preceding generation.

Acknowledgments. I am indebted to my colleague A.D. Falkoff for suggestions which greatly improved the organization of the paper, and to Professor Donald McIntyre for suggestions arising from his reading of a draft.

Appendix A. Summary of Notation

$F\omega$	SCALAR FUNCTIONS	$\alpha F\omega$	
ω	Conjugate	+ Plus	
$0-\omega$	Negative	- Minus	
$(\omega>0)-\omega<0$	Signum	× Times	
$1\div\omega$	Reciprocal	÷ Divide	
$\omega\lceil-\omega$	Magnitude	| Residue	$\omega-\alpha\times\omega\omega\div\alpha+\alpha=0$
Integer part	Floor	P Minimum	$(\omega\times\omega<\alpha)+\alpha\times\omega\geq\alpha$
$\lceil-\omega$	Ceiling	$\lceil$ Maximum	$-(-\alpha)--\omega$
$2.71828\ldots\star\omega$	Exponential	* Power	$\times/\omega\rho\alpha$
Inverse of *	Natural log	⍟ Logarithm	$(\circledast\omega)\div\circledast\alpha$
$\times/1+\iota\omega$	Factorial	! Binomial	$(!\omega)\div(!\alpha)\times!\omega-\alpha$
$3.14159\ldots\times\omega$	Pi times	○	

Boolean: $\vee$ $\veebar$ $\sim$ (and, or, not-and, not-or, not)
Relations: $<$ $\leq$ $=$ $\geq$ $>$ $\neq$ ($\alpha R\omega$ is 1 if relation R holds).

```
                Sec.  V←→2 3 5       M←→1 2 3
                Ref.                    4 5 6
Integers          1   ⍳5←→1 2 3 4 5
Shape             1   ⍴V←→3    ⍴M←→2 3    2 3⍴⍳6←→M    2⍴4←→4 4
Catenation        1   V,V←→2 3 5 2 3 5    M,M←→1 2 3 1 2 3
                                                  4 5 6 4 5 6
Ravel             1   ,M←→1 2 3 4 5 6
Indexing          1   V[3 1]←→5 2    M[2;2]←→5    M[2;]←→4 5 6
Compress          3   1 0 1/V←→2 5    0 1⌿M←→4 5 6
Take,Drop         1   2↑V←→2 3    ¯2↑V←→1↓V←→3 5
Reversal          1   ⌽V←→5 3 2
Rotate            1   2⌽V←→5 2 3    ¯2⌽V←→3 5 2
Transpose        1,4  ⍉ω reverses axes    α⍉ω permutes axes
Grade             3   ⍋3 2 6 2←→2 4 1 3    ⍒3 2 6 2←→3 1 2 4
Base value        1   10⊥V←→235    V⊥V←→50
  &inverse        1   10 10 10⊤235←→2 3 5    V⊤50←→2 3 5
Membership        3   V∈3←→0 1 0    V∈5 2←→1 0 1
Inverse          2,5  ⌹ω is matrix inverse    α⌹ω←→(⌹ω)+.×α
Reduction         1   +/V←→10    +/M←→6 15    +⌿M←→5 7 9
Scan              1   +\V←→2 5 10    +\M←→2 3⍴1 3 6 4 9 15
Inner prod        1   +.× is matrix product
Outer prod        1   0 3∘.+1 2 3←→M
Axis              1   F[I] applies F along axis I
```

Appendix B. Compiler from Direct to Canonical Form

This compiler has been adapted from [22, p.222]. It will not handle definitions which include ⍺ or : or ⍵ in quotes. It consists of the functions *FIX* and *F9*, and the character matrices *C9* and *A9*:

```
FIX
0⍴⎕FX F9 ⍞

D←F9 E;F;I;K
F←(,(E='⍵')∘.≠5↑1)/,E,(⍪4,⍴E)⍴' Y9 '
F←(,(F='⍺')∘.≠5↑1)/,F,(⍪4,⍴F)⍴' X9 '
F←1↓⍴D←(0,+/¯6,I)+(-(3×I)++\I←':'=F)⌽F,(⍪6,⍴F)⍴' '
D←3⌽C9[1+(1+'⍺'∈E),I,0;],⍉D[;1,(I←2+⍳F),2]
K←K+2×K<1⌽K←I∧K∈(>⍀1 0⌽'←⎕'∘.=E)/K←+\~I←E∈A9
F←(0,1+⍴E)⌈⍴D←D,(F,⍴E)⍴⍉0 ¯2↓K⌽' ',E,[1.5]';'
D←(F↑D),[1]F[2] '⍝',E
      C9                    A9
   Z9←                   012345678
 Y9Z9←                   9ABCDEFGH
 Y9Z9←X9                 IJKLMNOPQ
)/3→(0=1↑,               RSTUVWXYZ
   →0,0⍴Z9←              ABCDEFGHI
                         JKLMNOPQR
                         STUVWXYZ⎕
```

Example:

```
      FIX
FIB:Z,+/¯2↑Z←FIB⍵-1:⍵=1:1

      FIB 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

      ⎕CR'FIB'
Z9←FIB Y9;Z
→(0=1↑,Y9=1)/3
→0,0⍴Z9←1
Z9←Z,+/¯2↑Z←FIB Y9-1
⍝FIB:Z,+/¯2↑Z←FIB⍵-1:⍵=1:1
```


References

1. Boole, G. *An Investigation of the Laws of Thought*, Dover Publications, N.Y., 1951. Originally published in 1954 by Walton and Maberly, London and by MacMillan and Co., Cambridge. Also available in Volume II of the *Collected Logical Works of George Boole*, Open Court Publishing Co., La Salle, Illinois, 1916.
2. Cajori, F. *A History of Mathematical Notations*, Volume II, Open Court Publishing Co., La Salle, Illinois, 1929.
3. Falkoff, A.D., and Iverson, K.E. The Evolution of APL, *Proceedings of a Conference on the History of Programming Languages*, ACM SIGPLAN, 1978.
4. *APL Language*, Form No. GC26-3847-4, IBM Corporation.
5. Iverson, K.E. *Elementary Analysis*, APL Press, Pleasantville, N.Y., 1976.
6. Iverson, K.E. *Algebra: an algorithmic treatment*, APL Press, Pleasantville, N.Y., 1972.
7. Kerner, I.O. Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen, *Numerische Mathematik*, Vol. 8, 1966, pp. 290-294.
8. Beckenbach, E.F., ed. *Applied Combinatorial Mathematics*, John Wiley and Sons, New York, N.Y., 1964.
9. Tarjan, R.E, Testing Flow Graph Reducibility, *Journal of Computer and Systems Sciences*, Vol. 9 No. 3, Dec. 1974.
10. Spence, R. *Resistive Circuit Theory*, APL Press, Pleasantville, N.Y., 1972.
11. Iverson, K.E. *A Programming Language*, John Wiley and Sons, New York, N.Y., 1962.
12. Iverson, K.E. *An Introduction to APL for Scientists and Engineers*, APL Press, Pleasantville, N.Y.
13. Orth, D.L. *Calculus in a new key*, APL Press, Pleasantville, N.Y., 1976.
14. Apostol, T.M. *Mathematical Analysis*, Addison Wesley Publishing Co., Reading, Mass., 1957.
15. Jolley, L.B.W. *Summation of Series*, Dover Publications, N.Y.
16. Oldham, K.B., and Spanier, J. *The Fractional Calculus*, Academic Press, N.Y., 1974.
17. *APL Quote Quad*, Vol. 9, No. 4, June 1979, ACM STAPL.
18. Iverson, K.E., *Operators and Functions*, IBM Research Report RC 7091, 1978.
19. McDonnell, E.E., A Notation for the GCD and LCM Functions, *APL 75, Proceedings of an APL Conference*, ACM, 1975.
20. Iverson, K.E., Operators, *ACM Transactions on Programming Languages And Systems*, October 1979.
21. Blaauw, G.A., *Digital System Implementation*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
22. McIntyre, D.B., The Architectural Elegance of Crystals Made Clear by APL, *An APL Users Meeting*, I.P. Sharp Associates, Toronto, Canada, 1978.